```
#!/bin/bash
# Tip:
# If you're unsure of how a certain condition would evaluate,
#+ test it in an if- test.
echo
echo "Testing \"0\""
if [ 0 ] # zero
then
echo "0 is true."
else # Or else ...
echo "0 is false."
fi # 0 is true.
echo
echo "Testing \"1\""
if [ 1 ] # one
then
echo "1 is true."
else
echo "1 is false."
fi # 1 is true.
echo
echo "Testing \"-1\""
if [ -1 ] # minus one
then
echo "-1 is true."
else
echo "-1 is false."
fi # -1 is true.
echo
echo "Testing \"NULL\""
if [ ] # NULL (empty condition)
then
echo "NULL is true."
else
echo "NULL is false."
fi # NULL is false.
echo
echo "Testing \"xyz\""
if [ xyz ] # string
then
echo "Random string is true."
else
echo "Random string is false."
fi # Random string is true.
echo
```

bash.

copy&paste от brigante скромно го систематизирах, но някои текстове се повтарят:) нищо де тъкмо ще се затвърдяват знанията:) когато имам време ще го преработя и може би ще понапиша и аз нещичко:)
не съм тествал,проверявал написаното дали е вярно! всеки автор сам си отговаря и разчитам на познанията му в областта :)
...надявам се да няма възражения от авторите, за това че съм им копирал уроците :) ако има ми свирнете :)
...разбира се може всеки читател на книгата да добавя свой текст и така да стане една хубава голяяяяяяяяма ел. книжка:) ъплоудвайте завършения .pdf файл и работният .odt файл(преименувайте така .odt.pdf т.е. да завършва на .pdf или .txt ,за да може да го качите в сайта 'щото има определени file extensions)
brigante.sytes.net/upload/
Написаните статии са от: Венцислав Чочев, Недялко Войняговски(phrozencrew),
Inventive, cleaver
http://bg.wikipedia.org/wiki/Bash
http://www.linux-bg.org/cgi-bin/y/index.pl?page=article&id=advices&key=344082225

Bash или съкратеното от Bourne Again Shell е команден интерпретатор, използван за редица операции изпълнявани в конзолен режим или иначе казано в режим без графична среда. В системите, които са UNIX или Linux-базирани, командният интерпретатор изпълнява функцията на „преводач“ между потребителя и ядрото на операционната система, както и за много други неща. Един от първите такива интерпретатори е бил Bourne shell (sh), който се използва и до днес (ще покажа примерчета по-късно).
Друг широко разпространен интерпретатор е C shell (csh) и т.н.

Да пристъпим към съществото на приказката, но преди това няколко уточнения… По-надолу в статията ще използвам думата shell (шел за по-бързо) за „обвивката“ на потребителите. Има два, а при някои системи и по няколко вида shell-ове – потребителски и root shell(root – това е най-пълноправният потребител в системата). Тези шелове имат специфични обозначения, като например потребителският шел изглежда така “ ventsi@debian:~$ “ (завършва с $), а root шела така – „root@debian:~#“ (# накрая).

Всичко в Линукс е или файл или процес. Процеса е извикана програма/код, която се стартира с определен PID (идентификационно номерче). Файл е колекция или сбор от данни, които са написани от потребител използващ някакъв текстов редактор, компилатор и прочее. Всичко това се разпределя в директории, които съдържат информация за нещата в тях.

Цялата идея на горният абзац е да покаже, че в *nix (всичко, де що шава под Linux, Unix и други, ще съкращавам *nix) базираните системи използват така наречената „дървовидна структура“, т.е. всичко е на определени нива (директории и поддиректории, както е при DOS).

Когато един потребител отвори дадено приложение, той просто извиква написан код, който се изпълнява посредством изпратена команда към командният интерпретатор и т.н.

Както казах по-горе, всичко е подредено в „дървовидна структура“.

Нека да започнем с едно примерче. Например искаме да видим какво съдържа текущата директория, посредством конзолен терминал:

Командата е ls (идващо от английското „list“).

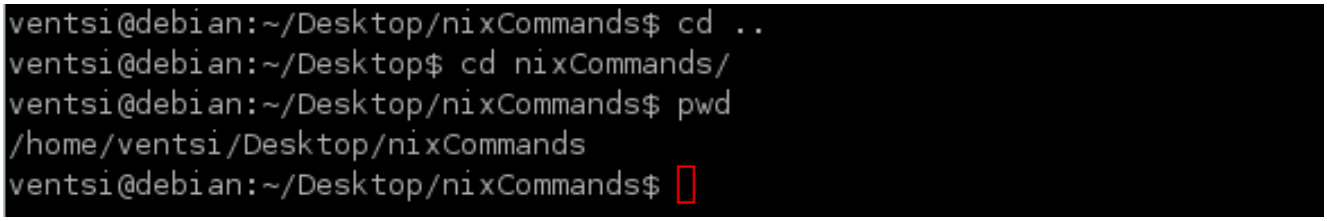
```
ventsi@debian:~/Desktop/nixCommands$ ls -lsa
total 8
4 drwxr-xr-x  2 ventsi ventsi 4096 2009-10-02 11:26 .
4 drwxr-xr-x 14 ventsi ventsi 4096 2009-10-02 11:26 ..
ventsi@debian:~/Desktop/nixCommands$
```

На картинката се вижда резултата. Както казах по-нагоре, всяка команда е задачка за операционната система, която е предварително дефинирана в различни файлчета или пакети (можем да ги редактираме, защото са OpenSource в превод „Отворен Код“). Когато напишем ls, ние извикваме скрипт/код с предварително зададени възможни опции (буквичките „lsa“ след “ – „). Всяка команда си има така наречените „man pages“ (по-късно ще разкажа и за тях), чрез които можем да разберем повече за дадена команда, нейните опции, създатели и прочее.
В случая ls -lsa казваме на ls командата, да върне всички файлове, подредени в йерархия, като първо са директориите, след това файловете и т.н. Повече можете да прочетете на вашата ситема след като изпълните man ls (отваря manual pages за дадена команда – излизането от man става с натискане на клавиш Q).

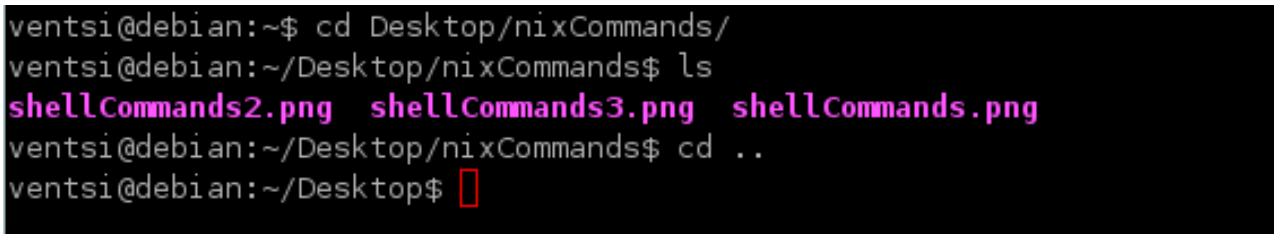
Върнатият резултат от командата е големина на файл, битове (дали е файл, кой може да го достъпва/права и други…), собственик, дата на създаване, час и име на файл с разширението му.

Както виждате има две директории (drwxr-xr-x, щом започва с d в първият символ, това значи, че е директория). Едната с име " . " е текущата директория. Другата с име " .. " е предишната в йерархията/дървовидната структура.

За да влезем в дадена директория с конзолата, правим така cd и името на директорията или пътят до нея (виж на картинката).



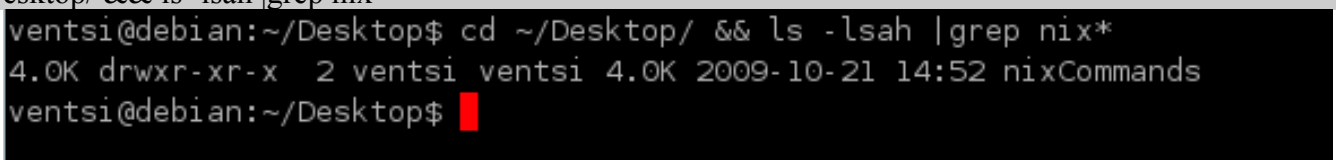
За да се върнем назад, т.е. в предната директория правим така cd .. (виж снимката)



Както се вижда на картинката, върнахме се от директория /home/ventsi/Desktop/nixCommands в предната/по-горната в йерархията, а в нашият случай това е /home/ventsi/Desktop .

Това до навигацията по директориите, останалото се учи в практиката, когато потрябва нещо по-сложно, например:

```
cd ~/Desktop/ && ls -lsah |grep nix*
```

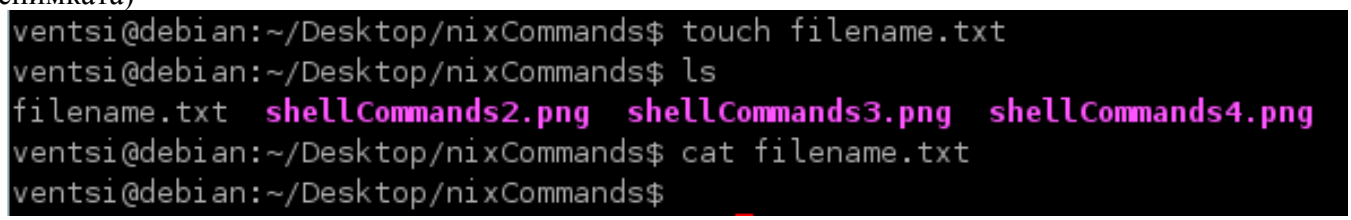


grep е приложение, което познава/търси даден текст или иначе казано нещо като много мощен филтър и не само (повече в описанието му – man grep).

Сега нека създадем малко файлчета в тази директория и да ги поманипулираме, какво ще кажете?

Създаване на файлче:

touch filename (това ще създаде файл с име filename), нека му зададем и разширение, просто за хубост touch filename.txt и да видим какво има в него посредством cat filename.txt
(виж снимката)



cat filename.txt не върна нищо, което значи, че файлчето е празно – разбира се, нали не сме го напълнили с нито един байт…

Нека вкараме малко текст в този файл:

```
echo „Това тук е малко текст в този файл.“ > filename.txt
```

Ето го и резултата на снимка:

```
ventsi@debian:~/Desktop/nixCommands$ echo "Това тук е малко текст в този файл." > filename.txt
ventsi@debian:~/Desktop/nixCommands$ cat filename.txt
Това тук е малко текст в този файл.
ventsi@debian:~/Desktop/nixCommands$
```

Това е просто малко примерче на работа с файлове. Сега нека да „хванем“ няколко думички в даден файл: (забележка: grep по подразбиране връща реда на който се намира даденият търсен символен низ…)

```
ventsi@debian:~/Desktop/nixCommands$ cat filename.txt | grep Това
Това тук е малко текст в този файл.
ventsi@debian:~/Desktop/nixCommands$ cat filename.txt | grep това
ventsi@debian:~/Desktop/nixCommands$
```

Както забелязвате на картинката – grep върна само един ред, защото толкова има в даденият файл, който съдържа думата „Това“, а на следващият опит не върна нищо, защото не намира думата „това“.

Инструменти, които улесняват нашата работа с Shell-а.

Тук темата е малко по-обширна и много мога да пиша, но ще направя кратко въведение и описание на инструменти, които помагат за приятна и удобна работа с конзолата.

Файлов редактор **vi/Vim**

Vim е подобрената версия на vi – файлови редактори, които са мощни, поддържащи голям набор от екстри и улеснения за потребителите.

(снимка на vim и нашето файлче от по-горе)

```
  1 Това тук е малко текст в този файл.
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12 Добавени празни редове
 13
 14
 15
 16
 17
```

MidnightCommander (mc)

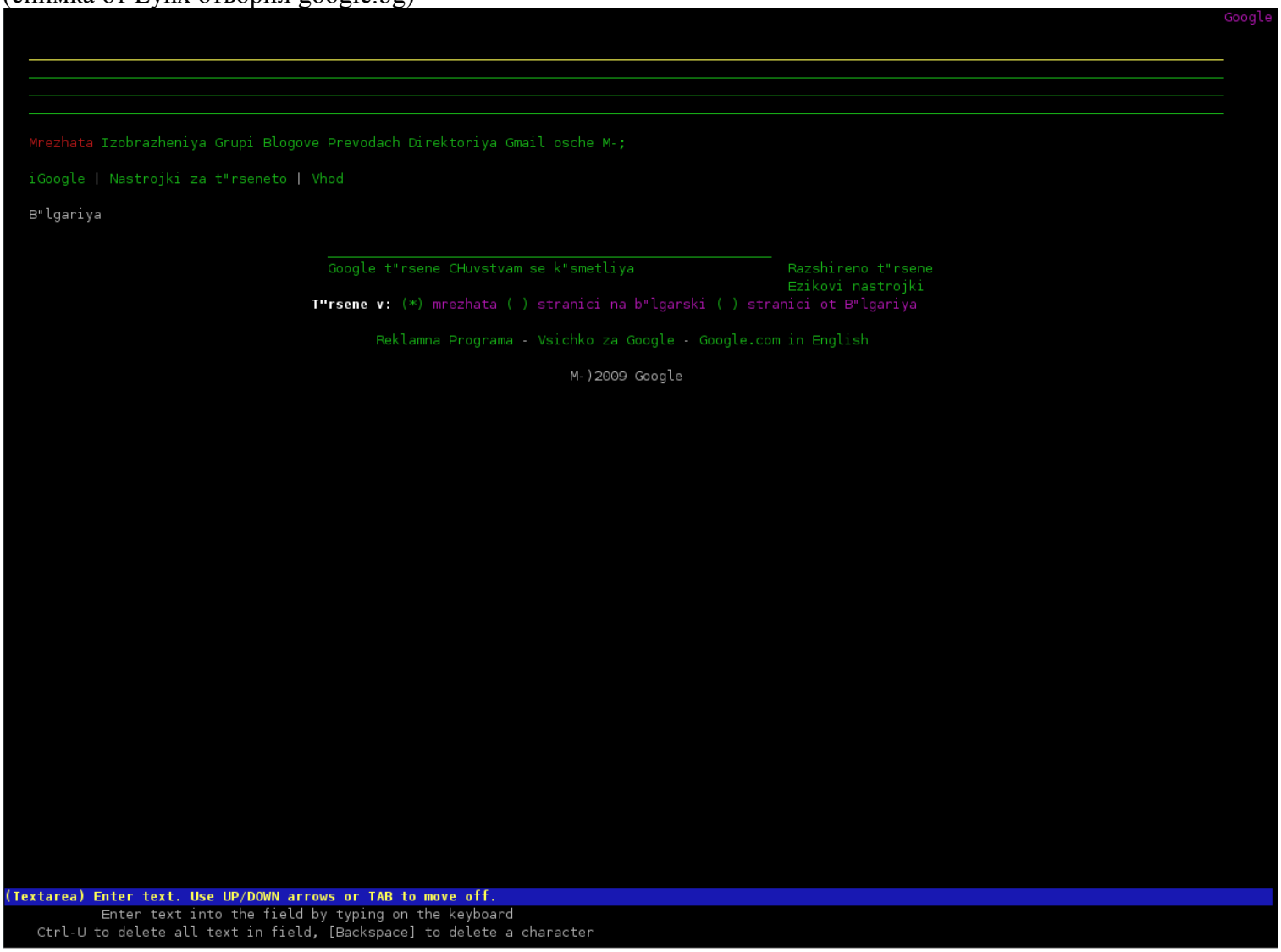
Файлов мениджър, който съдържа редица команди, които можем да напишем и на ръка, но определено е полесно да използваме интерфейс.

(снимка от mc)

Lynx – конзолен web браузър.

Да, наистина съществува и такова животно, което се справя с повечето добре направени сайтове. Е, разбира се, не може да подкара флаш, поне аз не знам метод, ако е възможна такава гимнастика, но се съмнявам дълбоко.
Браузърчето е удобно, ако на отдалечен сървър няма монитор и се достъпва от разстояние – за конзолата е страхотно направено, но си е hardcore, определено.

(снимка от Lynx отворил google.bg)



Венцислав Чочев
http://news.laptop.bg/статии/linux-bourne-again-shellbash-команди-или-как-да-работим-с-кон/

Bash (Bourne Again SHell) - приложно програмиране (от един любител) – v.01 (phrozencrew)

В този урок искам да разкажа за магнетизма на този популярен език, като разчитам за допълване от вас. Като един непрограмист бих нарекъл bash творчески или арт език, защото позволява използването на голямо богатство външни инструменти и дори най-простата задача може да се извърши по много начини. Всъщност истината е, че този език се е използвал от Шекспир за да напишее починал, преди да разкрие тайната на pipe-линията, която е използвал. Няма да се задълбочавам в техническите дълбини, а ще грабна леко от поетизма на езика, колкото да погъделичкам интереса на всеки, който е решил да използва свободните операционни системи. Този разказ ще е от един любител и фен на linux, без да претендир, но пичагата е починал, преди да разкрие тайната на pipe-линията, която е използвал. Няма да се задълбочавам в техническите дълбини, а ще грабна леко от поетизма на езика, колкото да погъделичкам интереса на всеки, който е решил да използва свободните операционни системи. Този разказ ще е от един любител и фен на linux, без да претендирам, че разбирам много от изкуството на програмирането с bash.ират могат да се съберат в една шепа, но всичко останало, с което може да взаимодейства е цял океан. Все пак ще се опитам да направя кратко съдържание на разказа. Това до тук е увод :)
Нека набележим някои базови точки:

1. Увод (е стига с тоя увод де!)
2. Променливи
3. Взимане на изход от програмаПредполагам е ясно, че bash е шел (конзола) по подразбиране на почти всички Linux/UNIX/Mac операционни системи. Добре е да се знае и че има още доста популярни шелове. А най-важното е да се знае, че в тези операционни системи всичко е файл! са на паркета, или ръчичките в тостера!!!)
9. test
10. Четене и писане (I/O) от конзолата
11. Специални символи - IFS, [ ], [[ ]], $#, ( ), (( )) ... Или как да се правим, че всичко е шифровано
12. Примерна програмка с база данни
13. Изводи и изход

Точките търпят промени и можете да добавяте всякакъв род предложения. Като започнем от сорта на "Общото между Bash и баш Бирата"

ПРОМЕНЛИВИ

Променливите в bash, както във всички останали езици са части от паметта, които можем да използваме в потока на програмата. Променливите се създават като напишем името на променлива, непосредствено следвана от знака "=" и след това и дадем стойност. Променливите се пишат в слят стринг (сбор от някакви символи), като е желателно да използвате латински букви и долна черта(максимум!). Когато променливата ви е съставена от няколко думи тя трябва да е затворена в единични или двойни кавички. За да видите променлива трябва да и сложите знак за стринг и да я изпълните с команд6. Използване на инструменти, препоръчителни за прохождащи поети на basho не ме целете с камъни, постарах се да е ясно. За пример нека запалим един терминал и запишем следните редове в него:

```
prom=Moeto
MyIP=127.127.127.127
quest=". Kefi6 li, a??"
```

За да видим какво безобразие сме съхранили в паметта на злощастната ни машина нека напишем следния ред:
Bash и баш Бирата
echo Променливите в bash, както във всички останали езици са части от паметта, които можем да използваме в потока на програмата. Променливите се създават като напишем името на променлива, непосредствено следвана от знакаот по-горната команда е:
Moeto IP е 127.127.127.127. Kefi6 li, a??
Ако сложим целия израз в общи кавички и го изпълним, ще получим същият резултат:

```
echo "$prom IP е $MyIP$quest"
```

Какво обаче ще се случи, ако сложим този израз в единични кавички:

```
echo '$prom IP e $MyIP$quest'
```

Резултата ще е, че ще получим същия отговор, като въпроса:
$prom IP e $MyIP$quest
Това е особено полезно, когато искаме да покажем даден променлива или специален символ, без да се изпълнява от bash. Разбира се това можем да го направим и по друг начин, като използваме специален ескейпващ символ "\":

```
echo "\$MyIP = $MyIP"
```

и разбира се резултата е: $MyIP = 127.127.127.127
Ами какво ще стане, ако искаме да отпечатаме повече символи след променливата, например, ако сме написали echo $MyIPto? Как да кажем на bash, че искаме да се ограничи само до символите на променливата? Интересен въпрос с много елегантен отговор. Bash ви съветва в този случай да сложите, след знака за стринг, променливата в ъглови скобки:

```
echo ${MyIP}to
```

резултат:
127.127.127.127to
В тези ъглови скоби можете да задавате и зелени карти (wildcards), които да ограничат каква част от присвоената променлива да се визуализират, но това е вече за много напреднали поети на bash, които са на повече бирички. Все пак ето малко доп.инфо. На този етап това ни е достатъчно за променливите. Но пък как можем да присвояваме стойности на променлива, които са върнати от изпълнение на дадена програма ...

ВЗИМАНЕ НА ИЗХОД ОТ ПРОГРАМА
Bash разполага с два основни начина за изпълнение на програми, като $(programa), `programa`. Но специалистите препоръчват $(programa). Как можем да присвоим изход на променлива от програма? Ще използвам една проста програма, която да ни покаже тази възможност - date. Програмата date връща отговор текущата дата, като самата команда разполага с възможност за форматиране на изхода от изпълнението и, т.е. можем да визуализираме точно тази част от датата, която искаме и то по такъв начин, който ни е удобен. В общия случай:
bash. Разбира се това можем да го направим и по друг начин, като използваме специален ескейпващ символог: #000066;">echo $datata

Резултата от изпълнението на командата е:
Mon Nov 16 21:09:44 FLEST 2009
Ако вкараме и малко форматираме, можем да получим чудИса :):

```
datata=$(date +"%d.%m.%y")
datata=`date +"%d.%m.%y"`
```

Което ще ни върне същият резултат.

Повечето конзолни инструменти в свободните операционни системи връщат резултат, който може да се присвои на променлива. Това е много важно да се знае. Като едни лаици бихме могли да преформатираме и изхода от изхода. Само за гъдел ще чопна единствено датата, която се визуализира от горния скрипт:

```
echo $den
```

Резултата от изпълнението на командата е:
(Ако още веднъж използвам по-горния израз ще си прегриза сухожилията на коляното!)
11
Хитро а? :) Нека кажем някои неща за този символ "|". Този символ се използва за навързване на няколко команди, като изхода на предишната се подава като вход на следващата. Това е една много мощна философия и религия от древните времена на UNIX. още от преди родителите ви да са били родени :), че дори преди да сте им били в любовния поглед :). Английското название на този вид навързвBash разполага с

два основни начина за изпълнение на програми, като $(programa), `programa`. Но специалистите препоръчват $(programa). Как можем да присвоим изход на променлива от програма? Ще използвам една проста програма, която да ни покаже тази възможност - date. Програмата date връща отговор текущата дата, като самата команда разполага с възможност за форматиране на изхода от изпълнението и, т.е. можем да визуализираме точно тази част от датата, която искаме и то по такъв начин, който ни е удобен. В общия случай: интересни свойства, които е редно да се отбележат в нашия туториал. Няма да разглеждам всички възможности, защото можете да използвате man echo, за да разберете повече.
Имайте в предвид, че има огромна разлика дали echo ще се използва с кавички или без кавички. Разликата е, че ако използвате кавички ще можете да използвате и някои специални символи, НО само ако добавите опцията -e към echo.
Основните възможности:
\a - издаване на звук (камбанка)
\c - подтискане на новия ред
\n - нов ред
\r - връщане на каретката (в MAC това е стандарт за нов ред)
\t - хоризонтална табулация
Примери:

```
$ echo "\n"
n
#
```

```
$ echo -n "\n"
\n
```

Това в случая е същото като:$ echo -e '\\n'"\c"
\n

```
$ echo -e "\n"
#нов ред
#н<
```

Ако обаче не добавите **-e** , тогава когато сложите в скоби някоя от по-горните опции, няма да се случи нищо особено, освен принтирането на опцията.
Ако не използвате кавички, ляво наклонената черта ще действа като ескейпващ символ и нищо повече.
Друго интересно е да се знае, че понякога може да се получи пренасяне на нови редове на принтирания в конзолата резултат. Вариант да се избегне това е като се използва опцията **-n** или **-e** с **\c** (в примерите по-горе). Това обикновено се случва, когато имате върнат резултат от изпълнението на друга програма.

Често за по-добра четимост се използва и echo с умишлено пренасяне на нов ред на текста. За да пренасяте използвайте ескейпващ символ "\". Специално за тази възможност няма значение дали текста е в двойни кавички или не.

```
$ echo Towa e edin mnogo \
dylyg teks
```

Резултат:
Towa e edin mnogo dylyg tekst.
Абе тариактска работа!

printf

Е, нашият разказ логично опря и до супер(хипер-мега) полезеният инструмент printf. Този инструмент е замислен като наследник на конфликтното echo от старите времена на UNIX. Проектиран е да създаде комфорт и дава мощен контрол над принтирането в командния ред (CLI-тарикатско име, нали! Като на извънземно :)).
Поетите на повечко бирички ще си помислят, че тука нещо има съвпадък. Ами дам, това е почти същият инструмент като функцията printf() в езика за програмиране C (ах този магьосник, де се е навряла пущината!).

Синтаксис на програмата printf:
Подробния синтаксис бих го записал по този начин:
printf [<форматира 

Добре (но не задължително) е да подадем равен брой форматиращи елементи и аргументи!
Типичния изглед е този:

```
ime=Baio
familia=Baiov
printf "Ime: %snFamilia: %s\n" "$ime" "$familia"
```

Резултат:
Ime: Baio
Familia: Baiov
Където, според синтаксиса имаме 2 форматиращи елемента (%s и %s) и два аргумента, подадени за форматиране ("$ime" и "$familia").
Въпроса, дали Байо е Зайо или е друго ньедно животно, си го задавайте сами преди лягане. И ако е байо тоя заьо, защо?! Най вече! Аз повече на провокации няма да отговарям!
Естествено, че някой опортюнист са ше каже, -Добре е бай ..ъ-ъ-ъ пичага, ами как ше принта знак процент, а? Лесно, майна! Просто слагате два броя от тая хава "%%".

```
procent=85
printf "Tuka malko procent4e: %s%%\n" "$procent"
```

Резултат:
Tuka malko procent4e: 85%

Нека дадем и малко обяснение за тия проценти, тия кавички и въобще. Системата е следната. Ако не зададете никакво форматиране, то тогава printf ще изпечата всичко, което го накарате и то почти буквално, но с условности. Тогава е добре да ползваме единични кавички, защото, ако използвате двойни, има шанс да настъпите някой специален символ, като "!" например. Затова:

```
printf 'Pe4atai bukvalno! '$procent%%
```

Резултат:
Pe4atai bukvalno! 85%
Излиза, че можем да ползваме printf като echo, но трябва да внимаваме със символите. За това пък си има едно %b, което казва на printf да работи като echo. Е това е добре, но какво стана с онова %s. Ще разгледам няколко основни форматиращи инструкции (ще го патентовам тоя израз - форматиращи инструкции, е гати авторитетното!).
%d - Принтираме целочислена десетична стойност.
%i - Това е синоним на по-горното. Прави същото.
%f - Принти стойности с плаваща запетая - 2.35, 0.12546...
%s - Принти стрингове, дет се вика прав текст.
%c - Принти символ
%b - Ей това е много хитро, защото позволява ескейпващи символи.
За другите магии си използвайте man printf.
Кои са основните ескейпващите символи, които можем да ползваме с %b? Същите, както и при echo:
\\ - Принти ляво наклонена черта "\"
\n - Принтира нов ред
\r - Принтира връщане на каретката
\t - Принти хоризонтален табулатор
Сега един пример:

```
printf "Cifri4ka: %dnString: %snPlavashta zapetaia: %f" 9 "Alooha banda" 3.14
```

Резултат:
Cifri4ka: 9
String: Alooha banda
Plavashta zapetaia: 3.140000

Има още нещо мнооого важно! При форматирането можете да задавате обхват, в който аргументите да се изпечатат. Да вземем на пример стринга "Alooha banda". Ако искаме да изпринтираме само първите 6 символа можем да зададем следния обхват %.6s:

```
printf "String: %.6s" "Alooha banda"
```

Резултат:
String: Alooha

Нека задълбаем още малко. Ако използвате този формат за числата с плаваща запетая (%f), тогава ще можете да се възползвате и от математическото закръгление на числата. Пример:

```
printf "Float: %.2f" 3.145
```

Това ще върне закръглението:
Float: 3.15

Останалото е история! Ех, че прозаично. Само дето пиша глупости. Останалото е много упражнения и лекинко попрочитане на man или info.

МАСИВИ - bash arrays

Както в останалите езици за програмиране и bash поддържа работата с масиви (arrays). С ограничението, че масивите са едномерни, поне за версиите до сега. Може би на някой няма да му е ясно какво точно представлява един масив, за това ще се опитам да обясня с метафора. Да вземем на пример един домат - ей това е масива :). И после го изяждаме. Е може и на салатка да си го нарежем, а с една тънка мастичка... малиии.
Майтапя се! Та да вземем за пример една пазарска чанта - масив. И започнем в нея да слагаме някакви продукти - елементи на масива. Така погледнато декларирането (някои поети биха използвали инициализиране) на този масив в bash би изглеждало така:

```
chanta=("birichka" "rakijka" "mastichka" "krastavichka")
```

NB! Масивите могат да се обявят без скоби, а и без кавички, но в скоби, ако са цели думи, но така е по-културно.
Само дето това си е пълна чанта с бая алкохол в нея, т.е. не я пълним, а си я поръчваме пълна. Абе днеска да не е празник нещо или е пятница? А честито!
Как ли можем да видим всичкия амбалаж, който сме сложили в нея, като използваме малко магия с ъглови скоби и други рунически символи:

```
echo ${chanta[@]}
```

Резултат:
birichka rakijka mastichka krastavichka

Същият резултат ще постигнем и с заклинанието ${chanta[*]}, т.е. ако сменим кльомбата с астерикс.
Разбрахме как да си вземем пълна чанта, и как да видим какво е сложено в нея, но да видим как можем да добавяме допълнителни елементи. В случая, с това хубаво пиене не можем да не сложим и малко маслинки в чантата. Добавянето на елемент към масива става по следния начин:

```
chanta=(${chanta[@]} maslinki)
```

Ако държим да сме мега-изрядни би трябвало да сложим и малко кавички, но за сега не е задължително. Все пак това ще е по-правилно:

```
chanta=("${chanta[@]}" "maslinki")
echo ${chanta[@]}
```

Резултат:
birichka rakijka mastichka krastavichka maslinki

До тук добре, обаче с тая ракийка ще я втасаме, та за това викам да махнем поне нея. Как се маха елемент от масив:

```
unset array[1]
```

Командата **unset** премахва елемент от масив, тя може да премахва и променливи, че дори и целият масив, стига да напишем заклинанието unset chanta.
Добре, но защо махнахме елемент едно, след като ракийката в списъка "chanta" е втора под ред? Естествено защото всяко броене в компютърния свят започва от нула "0". Затова първи номер в списъка е chanta[0]=birichka. Добре запомнете с кое да започнете купона :)!
Естествено има и друг начин да изтриете елемент от масив, просто като го приравните към нищо chanta[0]= .

Друго интересно нещо, което бихме искали да знаем е колко общият брой на продуктите в чантата, или елементите в масива. Това също става с едно простичко и логично заклинание ${#МАСИВ[@]} или ${#МАСИВ[*]}. В нашия случай ще изглежда така:

```
echo "Broi produkti w chantata: "${#chanta[@]}
```

Резултат:
Broi produkti w chantata: 4

Можем да правим и други магарии с масивите, например можем бързо да видим индексите на масива ни ${!chanta[*]}, което сега ще върне 0 1 2 3.

Мисля, че за сега това е напълно достатъчно знание за масивите, за да си направим поемата.

ИНСТРУМЕНТИ, КОИТО Е ДОБРЕ ДА ПОЗНАВАМЕ

Основните инструменти, които се използват с bash (и другите шелове) са събрани в един пакет наречен coreutils. Този пакет пък от своя страна се дели на още няколко секции - textutils, shellutils, fileutils и т.н. Тъй като това е съвкупнст от десетки инструменти, които едва ли ще ви се наложи да ползвате, ще се ограничa до семпло описание само на няколко по-популярни. Понякога може да ни се случи да забравим как точно се изписва дадена команда. Тогава за посещане можем да използваме клавиша TAB, като изпишем поне първата буква на команда и после натиснем TAB. Обикновено се натиска 2 пъти, но зависи от дистрибуцията, която ползваме. TAB ще ни върне всички инструменти, които започват с буквата или буквите, които сме изписали.

info

За да поучите информация за някой от инструментите в coreutils можете да използвате командата info coreutils следвана от името на инструмента:

```
info coreutils ls
```

Командата **info** изважда документацията за конкретен инструмент. Формата в който вади информацията позволява да се използва линкова система. По-горния пример ще изведе нещо подобно:

'ls': List directory contents
==========

The 'ls' program lists information abo...
...
* Menu:

* Which files are listed::
* What information is listed::
* Sorting the output::
* More details about version sort::

* General output formatting::
* Formatting file timestamps::
* Formatting the file names::
...

Тези редове, които започват с астерикс "*" са линкове и дават подробна информация за опциите с които разполага дадения инструмент. За да посетите някой линк идете с курсора до него и натиснете Enter. Превъртане на страницата надолу и нагоре се прави с клавишите "Page Down" и "Page Up". Отваряне на следваща страница - с клавиш "n", предишна страница - клавиш "p". Връщането на главната страница става с бутона "u".

ls

Накратко командата "**ls**" листва (извежда в списък) съдържанието на директориите. Ако искаме да листнем директория, която се намира някъде си, тогава пишем или абсолютният път, или релативният път до нея:

```
ls /absoliuten/pyt/
ls ../relativen/pyt
```

../ - това ни извежда една директория по-горе
./ - това ни дава текущата директория
/ - с товаМАСИВИ - bash arraysвата директорКакто в останалите езици за програмиране и bash поддържа работата с масиви (uot;/". Сравнението е много неподходящо, но не се сещам как може да се листне My Computer в Win.

chmod

Както споменахме в началото, всичко в Linux/UNIX е файл, независимо дали става дума за директория, хардуер или нещо друго. Затова след като се запознахме с ls нека видим още един много важен инструмент: chmod - сменя правата за достъп до файл. Това не се отнася до символните линкове! За да видим какви са правата за достъп до файловете в текущата директория можем да използваме:

```
ls -l
```

Резултат:
bash би изглеждало така:(d) или файл ("-"). След това символите се разглеждат по тройки. Първата тройка символи покзва правата върху файла на потребителя (собственика на файла). В случая имаме rwx, което означава, че потребителя може да:
r - чете файла
w - записва във файла
x - изпълнява файла

Втората тройка символи показва правата на групата, а третата тройка правата на всички останали.
Ако искаме да позволим изпълнението (execute) на някой файл от всички потребители, тогава можем да му сменим правата по този начин:

```
chmod +x file.sh
```

Ако го приложа за файла if.sh от по-горе, тогава ще имам следния резултат:
-rwx--x--x+ 1 bob users 680 Feb 17 2009 if.sh
Ако искаме да дадем право на групата да го чете, тогава трябва да направим следното:

```
chmod g+r if.sh
```

Както виждаме групата се отбелязва с "g". Да видим всички възможности:
u - Потребителят, собственика на файла.
g - Групата, към която принадлежи собственика на файла.
o - Останалите.

a - Всички.
По принцип, когато даваме права за всички, тогава няма нужда да пишем "a". За това и по горе написахме само +x.
Когато искаме да отнемем права, тогава използваме знака минус "-".

```
chmod a-x if.sh
```

Така направихме файла неизпълним за всички.
За директории също можем да използваме chmod. Ако искаме дадена директория да може да се листва само от потребителя и групата, тогава правим следното:

```
chmod a+r direktoria
chmod o-r direktoria
```

Това може да се направи още по-лесно, ако използваме цифровия формат на командата, но мисля, че няма нужда да усложняваме толкова урока.

cd

Тази команда ни помага да сменяме текущата директория и да се шляем насам-натам. Също, както и при "ls" и тук можем да сменяме директорията като използваме абсолютен или релативен път. Разполагаме и с възможност за телепортиране, като например:cd ~

Това ще ни закара директно в home директорията на потребителя с който сме се регнали в системата. Командата е същата, както и в Dos.

cat

Тази команда ще отпечата съдържанието на даден текстови файл:

```
cat text.file
```

Ако файла е по-голям и не се събира на монитора, тогава можем да използваме командата less в piping-режим:

```
cat text.file | less
```

less

Тази команда връща резултата от предишната команда страница по страница. С шпацията или PageDown можем да прелистваме следващата страница. С "q" прекратяваме изпълнението на less. Може да се използва за всеки интрумент, който връща списък или просто по-дълъг текст. Подобенен е на командата "more" в Dos.

```
ls ~/Desktop | less
```

Други интересни команди, свързани с правата за достъп и е добре да се познават поне бегло са - **groupadd**, **useradd**, **chown**.

sudo - Super User Do. Прави се от root-а (администратора). Команда, която ви дава права на супер потребител, който може да прави каквото си иска. Верно, че не съвсем понякога, щото нема да стане: sudo sipi-ena-rakia tuka :)

Още малко инфо за файловите команди:
mkdir - прави директории
rm - трие
cp - копира
move - мести от едно място на друго
touch - създава файл с аргумента на командата. Например: "touch me" ще създаде файла "me" в текущата директория.
pwd - полезно, показва ни къде се намираме в момента
vdir - това е "ls" на стероиди :)

Разни други команди:

wc - брои:
wc -W думи,
wc -l редове или
wc -m - букви във файл.
tac - същото като cat, но извежда файла или стандартният вход от долу нагоре
cal - календар. Ако искате да изкарате календар за цялата 2010-та: "cal 2010"

Мрежови команди
ifconfig - извежда информация за IP-то и параметрите на връзката (MAC-адрес и т.н.)
iwlist scan - сканира за мрежи, които са в обхвата на Wireless-a

du - Показва използване на харда от всеки файл и директория в текущата.
ps - извежда процесите. Пример "ps aux"
sort - инструмент за сортиране. Разполага с богат набор опции. Примери:
sort -n - сортира по числова стойност
sort -u - сортира и изкарва само уникалните редове
sort -r - реверсивно сортиране, на обратно
head - извежда първите 10 реда от файл или стандартен вход. Обикновено върви с параметър колко реда да изкара "head -5" - ще изкара 5 реда.
tail - извежда последните 10 реда от стандартния изход (от команда) или файл. Също върви с броя редове.
Пример с използването на двете команди head и tail (извеждане на 20-те най-големи директории в текущата):

```
du | sort -n | tac | head -20
du -xk | sort -n | tail -20
```

Търсене за файлове и директории
whereis - бързо намиране на файл
locate - бързо намиране на файл или директория
find - бавно намиране на файл :). Пример:

```
find . -name "*.txt"
```

lshw - Да видимОсновните инструменти, които се използват с bash (и другите шелове) са събрани в един пакет наречен coreutils. Този пакет пък от своя страна се дели на още няколко секции - textutils, shellutils, fileutils и т.н. Тъй като това е съвкупнст от десетки инструменти, които едва ли ще ви се наложи да ползвате, ще се ограничa до семпло описание само на няколко по-популярни. Понякога може да ни се случи да забравим как точно се изписва дадена команда. Тогава за посещане можем да използваме клавиша TAB, като изпишем поне първата буква на команда и после натиснем TAB. Обикновено се натиска 2 пъти, но зависи от дистрибуцията, която ползваме. TAB ще ни върне всички инструменти, които започват с буквата или буквите, които сме изписали.llspacing=0>

```
cdrecord -v dev=/dev/cdrom blank=fast
```

Рестартиране на системата:shutdown -r now

Спиране на системата:shutdown -h now

ЗАБАВА
apt-get moo - просто го пробвайте :)

-==[]==--

lsb_release -a - връща информация за дистрибуцията. Тази информация може да се види и по други начини. Например: cat /etc/issue, cat /etc/lsb-release. Командите са тествани под Ubuntu. Под RedHat може да се използва командата cat /etc/redhat-version.

uname -a - ще изведе информация за версията на кернела, типа ОС, датата и платформата - 32 или 64 битова. uptime - дава информация за времето в което машината е била включена, както и точния час в който е била включена.uptime
#Резултат: 19:23:35 up 1:14, 2 users, load average: 0.21, 0.0

wget - супер универсален доунлоадер с изключително много възможности. Може да се използва за сваляне на файлове, за дърпане на сорс на Web страници, създаване на огледален образ на отдалечена директория или сайт, сваляне на сайт за офлайн-браузване, по време на свалянето абсолютните линкове се конвертират в релативни и т.н. Примери:
Сваляне на HTML сорс кода на някоя страница:wget -q -O - download.bg

Сваляне на ISO-файл. Когато се свалят големи файлове има вероятност връзката да умре. Тогава за да продължим свалянето от там до където сме стигнали можем да използваме опцията -c (continue):wget -c http://nimue.fit.vutbr.cz/slax/SLAX-6.x/slax-6.1.2.iso

history - извежда списък с последните 500 команди, изпълнени в конзолата. Броя на командите, които са съхранени зависи от конфигурацията на bash. Ако искаме да намерим определена команда и си спомняме само част от нея, можем да използваме grep:history | grep "ls"

convert - конвертиране на изображения от един формат в друг:convert pic.png pic.jpg

prename - преименуване на файлове, като могат да се използва Perl регулярни изрази. Напоследък тази функционалност е добавена направо в инструмента rename.

sleep - изчакване за определено време. Много използвана команда. Формата е:
sleep число[smhd] s - секунди
m - минути
h - часове
d - дни
Тази команда ще накара bash да изчака 5 секунди преди да продължи:sleep 5s

expr - принти резултат от израз. Този израз може да бъде математически или логически. Позволява сравнява на стрингове.
Пример:
expr 3 != 2
Това ще върне 1 – истина.

Други примери за expr:

```
expr 1 + 3
expr 2 - 1
expr 10 / 2
expr 20 % 3
expr 10 \* 3
echo `expr 6 + 3`
```

seq - броене със стъпка (seq [от] [стъпка] [до]). Пример:seq 2 2 20

Ще брои от 2 до 20 със стъпка 2: 2, 4, 6, 8...
Тази команда често се ползва в цикли, където е много полезно стъпковото и броене.for i in `seq 1 10`; do echo $i; done;
#същото като
seq 1 10 | while read i; do ec

В някои дистрота seq е в компанията на jot ([от] [до] [стъпка]).

{a..z} - броене в обхват. Не се сещам как да го обясня по-правилно. Пример:echo {a..z}
#Резултат: a b c d e f g h i j k l m n o p q r s t u v

Подходящо е да се използва в комбинации с групи от символи. Да видим как ще получим всички възможни двусимволни комбинации от a до z и числата 1 и 2:echo {a..z}{1..2}

xargs - обработва аргументите подадени от предната команда. Може да се използва с обхватното броене. Пример:

```
echo {a..m}{a..m} | xargs -n1
```

Тази команда е много полезна и определено съветвам на нея да се обърне особено внимание.

tr
Съкращението на тази команда идва от translate. Можем да използваме командата като заместител, т.е. да заместим едни символи с други. Да видим как ще заместим големите букви с малки в някой стринг:

```
echo ALooHa | tr '[A-Z]' '[a-z]'
```

#Резултат:

или да заместим цифри с букви:

```
echo {1..4} | tr '[1-9]' '[a-z]'
```

#Резултат:

На командата може да се подаде файл със символа "<":tr '[A-Z]' '[a-z]' < file.txt

Можем да използваме и опцията "-d", която ще изтрие намерения шаблон. Да видим как можем да конвертираме DOS/Windows текстов файл до UNIX текстов файл:tr -d '\r' < dos_file.txt > unix_file.txt

cut
cut - отпечатва избрани части от стандартния вход. Тази команда може да работи в три режима, като най използваните са "-f" - обхватен и "-c" - символен. За да разбере, кои части от подадения ред да извлече, използва делиметър (разделител). Ще се опитам да покажа как функционира тази изключително полезна команда. Да разгледаме за начало по-елементарния начин на употреба – символния:

```
echo "123456789" | cut -c -5
```

#Това ще върне: 12345

echo "123456789" | cut -c 5-
#Това ще върне: 56789

echo "123456789" | cut -c 3-7
#Това ще върне: 34567

echo

При рязането на символи имаме точно определен шаблон, чрез който режем от стринга.

По интересен е обхватния режим на работа с командата - "-f". При този режим можем да използваме входящ и изходящ разделител "-d". Ако не използваме входящ разделител (делиметър), тогава ще имаме режим много подобен на символния от по-горе. Важно е да се знае, че при обхватния режим разделителя (делиметъра) по подразбиране е табулацията - "\t". Да видим един пример:
echo -e "1\t2\t3\t4\t5" | cut -f 3-
3 4 5

Тук режем всичко от третата част нататък.
Сега да видим и малко примери с входящ делиметър "-d":
echo "1|2|3|4|5" | cut -f 1,2 -d \|

Ако не поставим ескейпващия символ "\", ще влезем в pipe-режим, заради pipe-символа "|". За това и избрах толкова интересен пример. Разбира се, определено ще ни трябва да преформатираме изхода и да премахнем тази "тръба", за това ще добавим - "--output-delimiter=". На какво ще е равен този изходен разделител зависи от нашия избор.echo "1|2|3|4|5" | cut -f 1,2 -d \| --output-delimiter=,
#Резулта<

По подобен начин можем да видим и home директорията на всеки потребител на системата. В този случай ще използваме директно четене от файл ред-по-ред:cut –d: -f1,6 /etc/passwd
#Резултат: user1:/home/user1
#user2:

cut има и опция да показва само редове, които съдържат разделителя, като пропуска останалите. Това става с опцията "-s".

paste - поставя в линия два или повече файла. Какво имам в предвид. например имаме file1, който се състои от следните редове:1
2
3
4

Имаме и file2, който се състои от:edno
dve
tri

Ако използваме paste, за да свържем двата файла, ще получим следният резултат:paste file1 file2
1 edno
2 dve
3 tri
4
При тази команда, както и cut, можем да използваме делиметър (разделител):paste -d' ' file1 file2

read
С тази команда можем да четем. Обикновено четем данни въведени от потребителя:echo -n "Vasheto ime :"; read ime; echo "Priatno mi e $ime!"

read се използва често и в командите за циклене, с които ще се срещнем на по биричка с картофки след малко. Ето едно примерче за стимулиране на стомашно-чревния тракт :):info cat > cat.info
while read line;do nl; echo -e $line"\\n"; done < cat.info

В резултат имаме номерирани редовете от помощната информация за командата cat. Да обясним важното: while read line; do ... done < cat.info - Това преведено на нашенски значи "Докато четеш всеки ред от стандартния вход < cat.info, вземи да присвоиш всеки ред на променливата line".

Работа със стандартния изход и грешки.

При работа със конзолата често ще се налага да записваме във файл информация от изпълнението на дадена команда. Тази информация се нарича стандартен изход. Когато се получи грешка в изпълнението на даден скрипт или команда можем да запишем във файл и стандартния изход за грешки. Тези 2 типа изход се отбелязват и използват с определени символи. Стандартния изход е ">" или "1>".Стандартния изход за грешки е "2>". Понякога се налага да обединим стандартния изход и стандартния изход за грешки в един файл - "2>&1". Това обединяване пък често се използва за да съхраним информацията в нищото и да получим само потвърждение дали командата е успешна или не:info cat > cat.info 2> /dev/null

Стандартния вход е много лесен за ползване: "< file"

grep
grep - инструмент за търсене на съвпадение във всеки файл или в стандартния вход. МНОГО, МНОГО ВАЖЕН ИНСТРУМЕНТ. Формата на тази блестяща команда може да се изрази по този начин:grep [опция] "дума-за-на

Някои популярни опции при използване на **grep** са:
-v - избира несъвпадащи редове
-P - ШАБЛОНА е Perl регулярен израз
-i - игнорира малки или големи букви
-n - показва реда на който се среща шаб- извежда списък с последните 500 команди, изпълнени в конзолата. Броя на командите, които са съхранени зависи от конфигурацията на bash. Ако искаме да намерим определена команда и си спомняме само част от нея, можем да използваме grep:можностите му се разширяват от вградената разпознаване на WildCards (зелени карти), RegEx - Regular Expressions (регулярни изрази), POSIX регулярни изрази, стандартно търсене на текстов стринг. Да се напише всичко за тази команда ще са необходими много думи и примери. А изучаването и изисква продължително тестване, за да се постигне прилично ниво на владеене на командата. Все пак ще се опитам да покажа някои по-стандартни техники, за да си пийвате спокойно биричката и да работите с grep, така че да скътате работата.
Много разпространено приложение е извличане на синтезирана информация за други команди. Това ще го приложа върху хелпа на grep. Например искаме да се подсетим коя е опцията за да използваме командата с Perl съвместими регулярни изрази (PCRE):grep --help | grep Perl
#Резултат: -P, --perl-regexp PATTERN is a Perl regular exp

Това е много основно и полезно приложение. Можем да ползваме този директен подход и за да убием някой процес, който сТази команда ще накара bash да изчака 5 секунди преди да продължи: всички bash процеси:ps aux | grep bash

Да видим кои процеси използват най-много процесорно време. За тази цел ще използваме командния език awk и 2 регулярни израза "^" и "[^..]":ps aux | awk '{print $3"\t"$11}' | grep ^[^0]

Можем да използваме grep и за да си припомним команди, подадени в конзолата:history | grep "grep"

history | grep -i ps

Да видим кой е с права на root за системата:grep root /etc/passwd

Да видим обаче един много интересен пример. Да се опитаме да извлечем всички ъпдейтнати програми от началната страница на download.bg - хитро а?!wget -q -O - http://download.bg | grep -o -P '=12\>[0-9a-zA-Z .]*' | cut -d'>' -f2
#Резултат: Addax P2P 3.6.1.0
#Internet Cyclone 1.99
#twhirl 0.9.4
#The Form Letter Machine 1.11.01
#DOFUS 1.29
#PowerDVD 9.0.2201.0

Да вкараме малко обяснение на това, което направихме:
1. Сваляме с wget страницата и принтираме HTML-сорса на стандартния изход
wget -q -O - http://download.bg
2. Задаваме с пайп "|" на grep да обработи стандартния вход, като му задаваме опция "-o" да принтира само отрязъка на шаблона, а не целия ред. Задаваме му да използва и Perl-съвместими регулярни изрази "-P". След изграждаме самия регулярен израз, което правим след като сме разгледали сорс кода на index страницата на http://download.bg/. Там виждаме, че изброяването на всички ъпдейтнати програми става с:<img src='img/upd.gif' border=0 width=22 style='margin-right:3px;' height=12>Addax P2P 3.6.1.0

Чудесно и тъкмо за нас :). Един бърз сърч ни дава, че "height=12" се среща само в секцията с ъпдейтнатите програми. Следван от името на програмата, случая "Addax P2P 3.6.1.0" и най-накрая "". Преглеждаме програмит и виждаме, че са съставени само от букви, цифри, точки и шпации. Задаваме регулярния израз да търси за тези символи - [0-9a-zA-Z .]. Задаваме с астерикс "*" да търси за "0" или повече съвпадения на символите.
3. С cut слагаме делиметър ">" и извличаме само втората част на стринга - т.е. името на програмата:
cut -d'>' -f2
Ако сме майстори-поети и сме на 2-3 бирички, по-горния скрипт бихме могли да го оптимизираме така:wget -qO - http://download.bg | grep -oP '2\>[\d\w .]*' | cut -d'>' -f2

Интересни примери с grep могат да се видят на много места в мрежата. Можете да потърсите с "grep examples", "grep tutorials" или нещо подобно.

ЗАБАВА
Пробвайте една забавна игра с думи:
sudo apt-get install fortune-mod
echo;fortune;echo

Сега по необясними причини ще изпреваря планирането и ще ви запозная с...

sed

sed е поредния култов инструмент в нашата статия. Толкова е култов, че чак да ти се схванат глезените! По култов е и от "луннта диета за отслабване"! Е не е по-култов от биричката, обаче да пишеш за sed и да си мъцкаш биричка си е мега-култово.
По същество това е редактор за поточен текст. Обаче тоя редактор разполага с възможност за използване на регулярни изрази (Regular Expressions или RegEx). Ще разгледаме някои приложения на sed, макар, че е най-добре да съчетаем примерите с циклите. Но за цикленето по-късно.
Най-известната команда за sed е субституцията(замяната), която се използва с оператора "s":

```
echo den | sed s/den/nosht/
```

Този израз ще замести деня (den) с нощ (nosht).
Но какво значи поточен (стрийм) редактор, ако не можем да го приложим за файлове?! Както казахме по-

горе, за да вмъкнем за обработка текстов файл, използваме знака "<" (стандартен вход - stdin), а за да запишем резултата от обработката от конзолна команда в текстов файл използваме знака ">" (стандартен изход - stdout). В този случай бихме могли да използваме този UNIX синтаксиса <old >new за да заменим символите от един файл и да запишем промените в друг файл. Прилагайки по горния израз за деня, върху файлове, ще получим:sed s/Deniat/Noshtta/ <old >new
#Файла old е с съдържание:
#Deniat nastypva s purpurna voiska.
#Резултата е , че файла new съдържа:
#No

Вероятно се досетихте, че синтаксиса на sed, в тези случаи е:
sed [оператор]/оригинал/замяна/
Като оригиналния стринг и шаблона за замяна са разделени от дясно наклонена черта "/". Избора на разделите е абсолютно произволен! Можете да използвате и "щрихт но морн" :) (Пуцай Куме!) или пък кльомба @ (абе кой я измисли тая дума?), или пък тръбите, дето толкова ги говорихме "|". Все тая е какво ще използвате за разделите на изразите, важното е да предпазите замяната от разпознаване на символ, за разделяне! Ако използвате "/", тогава в шаблона за замяна трябва да укажете на sed да приеме този символ като част от стринга. Това се прави с ескейпващ символ "\":echo delimo/delitel | sed 's/delimo/delitel/del/delit/'

В този случай поставете целия израз на sed в единични или двойни кавички. Рядко ще се наложи използването на наклонени черти, но все пак е интересно да го има като вариант.

Да боцнем 2 маслинки с глътката мастичка и да разгледаме по-интересните възможности на мистър sed.

- Използване на & като съвпадащ стринг
Понякога се налага да добавим допълнителни символи към търсения текст. Например да поставим скоби на текста, който ни интересува. По стандартния начин бихме могли, ако знаем точно кой текст търсим, да използваме директна замяна, без регулярен израз:
sed 's/nosht/(nosht)/'
Но това не би ни свършило работа за по-сложни изрази, за това хитрците са измислили добавянето на специалния символ "&". Малко абстрактно звучи, за това да покажем един пример:sed 's/[a-z]*/(&)/'

В този случай съдържанието на стринга, който съвпада с търсенето се заменя от "&". И още нещо още по-яко. Това заместване може да се прави многократно:echo 'nosht 1253' | sed 's/[a-z]*/& &/'
#това ще върне:
#nos

Регулярният израз [a-z] замества цялата латинска азбука с малки букви - от "a" до "z". Знакът астерикс "*" казва на регулярната машина да търси за нула или повече елементи за замяна.

Опиянени нека видим и малко от хелпа на sed-иструктършънопредатора (ба*ти думата, много съм горд от нея):
Sed редактиращи команди
a\ Добави текст под текущата линия.
c\ Промени текста от текущата линия.
d Изтрий текста.
i\ Добави текст над текущата линия.
p Принтирай текста до стандартния изход.
r Прочети файл.
s Търсене и замяна на текст. Субституция.
w Запиши до файл.
Sed опции
-e SCRIPT Добави командите в СКРИПТ-а до набора от команди, които обработват входния текст.
-f Добави командите в СКРИПТ-файла до набора от команди, които обработват входния текст.
-n Заглушен режим.
-V Покажи версията.

Да забием само за адреналина и един интересен пример, който обработва изхода на ls:ls -l | sed s/^-[-rwx]*/Файл:/ | sed s/^d[-rwx]*

[AWK - ДА СЕ ДОПИШЕ ТУК]

Малко прескочихме нещо много тежко и наслаждаващо, обаче когато пием биричка ни се искат по-соленки неща. А пъкawk си е за една убава мстичка на кристали.. е може и ириш виски, ам със сигурност трябват по-сериозни питиета за да се схване. И така ...

if ... else ... fi и малко elif

Кратко обяснение на този цикъл. Ако условието е вярно, тогава изпълни команда1, в противен случай изпълни команда1.
Синтаксис:
if условие
тогава [then] Ако условието е истина
изпълни всички команди до else
в противен случай [else] Ако условието не е истина
изпълни всички конанди до fi
fi

И един пример. Напишете в gedit или друг редактор следния скрипт:#!/bin/sh
#
Скрипт, който ще ни покаже дали аргументите са позитивни или негативни
if [$# -eq 0]
then
echo http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html
С по-горния код показахме и много интелигентен начин за изход от приложение - exit 1. По дефиниция всички успешни команди в linux завършват с изход "0". Когато едно приложение или функция върне резултат различен от "0", тогава изхода от изпълнението е неуспешен. Linux много добре разчита изхода от дадена команда дали е успешен или неуспешен. Това е и основата за piping скриптирането.
Ъ-ъ-ъъъ, а дали не се повтарям...
Запишете скрипта като isnump_n, стартирайте конзолата и направете файла изпълним по следният начин:chmod 755 isnump_n
или
chmod +x isn

Не забравяйте, че прилагането на команди върху файл изисква да сте в текущата директория на файла (cd ...) или да посочите абсолютния или релативен път до него. Стартирането на скрипта става примерно така:isnump_n 23

В този цикъл можете да вграждате още if-else-fi вътрешни цикли. Това е за домашна биричка.
По-интересно ще е да разгледаме друга опция на цикъла. Опцията elif. Тя добавя една много лесно постижима мулти-условна система... Лелиии-и-и-и! Споко, много просто е!
Синтаксиса в случая е следния:

```
if [ условие1 ]
then
	...
		изпълнение на някакъв код
elif [ условие2 ]
then
	...
		изпълнение на друг код
elif [ условие3 ]
then
		...
		изпълнение на трети код
else
```

```
        ...
    изпълнение на краен код
fi
```

В примера с разпознаването на числата не бяхме включили варианта "0". Сега, когато разполагаме с elif можем много лесно да го добавим.#!/bin/sh
#
Скрипт, който ще ни покаже дали аргумента е позитивен, негативен или "0"
if [$# -eq 0]
then
echo "$0 : Вие трябва да напишете някакво целочислена число"
exit 1
fi

if test $1 -gt 0
then
        echo "$1 е позитивно"
elif [$1 -eq 0]

Важно е да се запомни, че дали ще ползвате инструкцията test или ъглови скоби е все едно!
За да обогатим знанията си за if нека видим и още един пример. Да напишем програма, която ще изтрие файл, който и посочим:#!/bin/bash
echo -n "Име на файл за изтриване : "
read myfile
echo -n "Сигурен ли сте (напишете yes или no) ? "
read confirmation
confirmation="$(echo ${confirmation} | tr ‘A-Z’ ‘a-z’)"
if ["$confirmation" == "yes"]; then"дума-за-hayle="color: #66cc66;">[-f $myfile] && /bin/rm $myfile || echo

За да изпълните програмата, не забравяйте, че трябва да я направите изпълнима с chmod +x или 755. Иначе няма значение какво име и разширение ще и дадете.
Стига с толкова ифове, че да не станат с цвят :)... е на тоя странен хумор и аз си се чудя, да ме извинявате!

Следва продължение ... awk, for, pipeline, стартиране на команди в поток и условия

Това е много основно и полезно приложение. Можем да ползваме този директен подход и за да убием някой процес, който според нас прави гадости. Например искаме да намерим кой е PID-а на всички bash процеси:ps aux | grep bashhttp://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html#!/bin/bash
... от phrozencrew на 16.11.09 22:11
http://new.download.bg/index.php?cls=forum&mtd=thread&t=224992&q=bash

От Inventive / 05.02.2005 12:58 http://mytech.bg/uroci/22/Други/319/Bash+Scripting+Tutorial
Bash Scripting Tutorial

Съдържание

[--- **Увод** ---]
Всички UNIX системи поддържат голям брой програмни езици, позволяващи на потребителите да си правят свои собсвени програми. Всички дистрибуции имат поддръжка за програмиране за множество командни интерпретатори, езици от по-високо ниво като Perl и Tcl/TK и GUI програмиране за графични среди като KDE и GNOME.
Най-често използваните обвивки (shells) са:

bash - bourne again shell
sh - shell
csh - C shell
tcsh - Tenex C shell (C shell без поддръжка на tab-completion)
zsh - Z shell
ksh - Korn Shell

Можете да прочетете повече за всяка от тях в техните manual pages. В този tutorial ще се занимаваме с програмиране за bash. Една програма за bash комбинира UNIX командите по даден начин, за да изпълни определена задача. Програмата за bash представлява текстов файл, съдържащ команди, които можете да въведете със най-обикновен текстов редактор като pico, vi и др. Този текстов файл се нарича скрипт за обвивката.
##

[--- Hello, World script ---]

Отворете любимия си текстов редактор и въведете

```
#!/bin/bash
echo "Hello, World"
```

за да направите традиционния Hello, World script. Можете да стартирате скрипта като напишете точка (.) пред името на файла. Ако файлът се казва hello, командата би изглеждала така:

```
$ . hello
Hello, World
$
```

Другият начин е да направите скрипта изпълним:

```
$ chmod +x hello
```

и да го стартирате с:

```
$ ./hello
```

```
Hello, World
$
```

По-нататък в tutorial-а ще използваме втория начин.
Скриптът hello съдържа два реда. Първия ред казва на системата коя програма да ползва, за да прочете командите във файла, а втория изкарва на екрана Hello, World.
Можете да въвеждате коментари в скриптовете си чрез знака "#".

```
#!/bin/bash
# Това е коментар
echo "Hello, World" # Това също
```

##

[--- **Променливи** ---]

За да присвоите някаква стойност на променлива трябва да използвате оператора за присвояване (знака за равенство (=)). Въведете името на променливата, знака за равенство и след това стойността й. Пример:

```
num=10
```

Обърнете внимание, че не трябва да използвате интервали около оператора за присвояване. Ако сложите интервал след него (num= 10), bash ще се опита да изпълни команда "num=" с аргумент 10.
Можете да се обърнете към стойността на променлива чрез знака за долар ($). Ето как би изглеждал скриптът Hello, World ако използваме променливи:

```
#!/bin/bash

var="Hello, World"
echo $var
```

Въпреки че стойностите на променливите могат да бъдат произволни символи, ако включите символи, които се използват от обвивката, възникват проблеми. Тези символи са: интервал, точка (.), знак за долар ($), по-голямо (>) и по-малко (<), (|), (&), (*), ({) и (}). Тези знаци могат да се използват като стойност на променлива ако са поставени в двойни или единични кавички или с обратно наклонени черти. С двойни кавички можете да използвате всички запазени знаци без знака за долар ($). За да го включите в стойността на променливата можете да използвате обратно наклонена черта преди него ($) или да оградите стойността на променливата с единични кавички. Следващия пример илюстрира всичко описано до сега.

```
#!/bin/bash

num1=10
num2=20
msg1="$num1 < $num2 & $num2 > $num1" # Тук ще се покажат всички знаци без знака за долар,
                        # който използваме за да се обърнем към стойността на
                        # променливата.
echo $msg1

msg2="$100 > $10" # Тук използваме обратно наклонена черта за да покажем знака за долар
echo $msg2

msg3='Here we can use all of these symbols: ".", ">", "<", "|", "$", etc'
# по този начин всички знаци ще се покажат на екрана, единственият символ, който не може да
# се използва е единична кавичка (')
echo $msg3
# При всички примери по-горе не е нужно стрингът да се слага в променлива.
```

```
echo ""$msg1" is not needed to print '$num1 < $num2 & $num2 > $num1'"
```

Изпълнението на скрипта:

```
$ ./primer1
10 < 20 & 20 > 10
$100 > $10
Here we can use all of these symbols: ".", ">", "<", "|", "$", etc
$msg1 is not needed to print '10 < 20 & 20 > 10'
$
```

За да присвоите на някоя променлива резултата от изпълнението на някаква команда, трябва да оградите командата в обратно наклонени апострофи.

```
#!/bin/bash

lsvar=`ls ~`
echo $lsvar
```

Изпълнението на скрипта:

```
$ ./primer2
books docs ircd movies source work phun stuff
$
```

Ако поставите команда в единични кавички след това можете да използвате името на тази променлива като друго име на командата.

```
#!/bin/bash

lssrc=`ls ~/source/c`
$lssrc
```

Изпълнението на скрипта:

```
$ ./primer3
me0w.c m00.c
$
```

За да използвате резултата от изпълнението на команда в стринг или променлива трябва да заградите командата в скоби и да поставите знака за долар пред нея ($(ls)).

```
#!/bin/bash

echo "The date is $(date)"
```

Изпълнението на скрипта:

```
$ ./primer4
The date is пн юни 10 20:36:49 UTC 2002
$
```

В bash има и няколко специални променливи, които се използват за аргументите на скрипта за обвивката.

$0 - име на команда
$1 до $9 - аргументите на скрипта
$* - всички аргументи от командния ред
$@ - всички аргументи от командния ред поотделно

$# - броят на аргументите от командния ред

Ще разберете разликата между $* и $@ след като прочетете за контролната структура for-in. Погледнете долния пример за по-голяма яснота за специалните променливи.

```
#!/bin/bash

echo "The first argument is $1, the second $2"
echo "All arguments you entered: $*"
echo "There are $# arguments"
```

Изпълнението на скрипта:

```
$ ./primer5 arg1 arg2 arg3 arg4
The first argument is arg1, the second arg2
All arguments you entered: arg1 arg2 arg3 arg4
There are 4 arguments
$
```

Ако изпълните друг скрипт от текущо изпълняващият се, текущият скрипт спира изпълнението си и предава контрола на другия. След изпълнението му се продължава първия скрипт. При този случай всички променливи,
дефинирани в първия скрипт не могат да се използват във втория. Но ако експортирате променливите от първия те ще могат да се използват във втория. Това става с командата export.

```
#!/bin/bash

# Това е първият файл.

var=100
export var
./primer6-a

# EOF


#!/bin/bash

# Вторият файл.

echo "The value of var is $var"

# EOF
```

Изпълнението на скрипта (обърнете внимание че вторият файл трябва да е primer6-a, ако използвате друго име, сменете името на файла в първия скрипт).

```
$ ./primer6
The value of var is 100
$
```

Друг начин за деклариране на променливи е командата declare. Синтаксисът на тази команда е:

declare -тип име-на-променливата

Типовете променливи са:

-r - readonly
-i - integer (цяло число)
-a - array (масив)
-x - export

Пример:

```
#!/bin/bash

declare -i var # декларираме променлива от тип integer
var=100
echo $var
declare -r var2=123.456 # декларираме readonly променлива
echo $var2
var2=121.343 # опитваме се да променим стойността на var2
echo $var2   # стойността на var2 ще е все още 123.456
```

Изпълнението на скрипта:

```
$ ./primer7
100
123.456
bash: var2: readonly variable
123.456
$
```

Повече информация можете да видите в bash manual pages (man bash).

##

[--- **User input, <<, аритметични операции** ---]

За да четете входни данни от вашия script трябва да използвате командата read. В нея няма нищо трудно, но все пак вижте примера:

```
#!/bin/bash

echo -n "Enter a string: "
read str
echo "String you entered: $str"
```

Изпълнението на скрипта:

```
$ ./primer8
Enter a string: hello
String you entered: hello
$
```

С оператора << можете да предавате данни на някоя команда. След него трябва да поставите свой ограничител,
който представлява някаква дума (пр. EOF) и след данните трябва да поставите същата дума.

Пример:

```
#!/bin/bash

cat << EOF
sth
data
line
EOF
```

Изпълнението на скрипта:

```
$ ./primer9
sth
data
line
$
```

Командата **let** в bash се използва за **аритметични операции**.
С нея можете да сравните две стойности или да
извършвате различни операции като събиране и умножение.
Тези операции често се използват за управление на
контролни структури (например цикъла for, за който ще прочетете по-нататък) или за извършване на изчисления.
Синтаксисът на командата е показан по-долу:

```
$ let value1 operator value2
```

Освен този синтаксис можете да използвате двойни скоби.

```
$ (( value1 operator value2 ))
```

Можете да използвате направо операторите за аритметични операции ако и двете променливи са от един и същи
тип (например integer). Пример:

```
#!/bin/bash

echo -n "Enter the first number: "
read var1
echo -n "Enter the second: "
read var2
declare -i var3

echo ----------
echo "$var1 + $var2 = $(( $var1+$var2 ))" # тук използваме двойни скоби, както виждате трябва да сложим $ пред
                                          # израза, за да се изчисли
let res=$var1*var2
echo "$var1 * $var2 = $res"
var3=100
var3=$var3+10
echo "$var3" # тъй като тази променлива е декларирана като integer не е нужно да използваме командата let
```

Изпълнението на скрипта:

```
$ ./primer10
   Enter the first number: 10
   Enter the second: 3
   10 + 3 = 13
   10 * 3 = 30
   110
   $
```

Можете да използвате всеки от изброените по-долу оператори с командата let.

```
   +    - събиране
   -    - изваждане
   *    - умножение
   /    - деление
   %    - остатък при деление
   >    - по-голямо
   <    - по-малко
   >=    - по-голяма или равно
   <=    - по-малко или равно
   ==    - равно
   !=    - не е равно
   &    - логическо И (AND)
   |    - логическо ИЛИ (OR)
   !    - логическо НЕ (NOT)
```

За аритметични операции и сравняване можете да използвате командата expr, но тук няма да я описвам. За повече
информация вижте нейната документация. За да правите по-точни изчисления използвайте езика bc (http://www.gnu.org/software/bc/bc.html).

##

[--- **Условия** ---]

В тази част ще се запознаем с контролните структури за условия. Много от тях са подобни на структурите в другите
езици, но въпреки това има някои разлики. Условията често изискват да се изпълни някаква проверка, която сравнява
две стойности. Тази проверка се извършва чрез командата test. Синтаксисът на командата е показан тук:

```
   $ test value1 -option value2
   $ test string operator string
```

Обърнете внимание, че при сравняване на стрингове се използва оператор, а не опция. Вместо командата test можете
да използвате квадратни скоби ([и]). Командата test $var -eq 1 може да се запише така:

```
   $ [ $var -eq 1 ]
```

Резултатът от командата test се запазва в специалната променлива $?. Ако той е true, то $? е равна на нула, ако е
false то $? е равна на едно. (В променливата $? всъщност се съхранява изходния код на програмата, която е изпълнена.
Ако тя е завършила успешно то той е 0, ако не, той е друго число. При командата test, ако резултатът е true тя спира
с изходен код 0, ако е false с 1)

Пример:

```
#!/bin/bash

var1=10
var2="m000"
[ $var1 -eq 10 ]
echo $?
[ $var2 = "me0w" ]
echo $?
```

Изпълнението на скрипта:

```
$ ./primer11
0
1
$
```

Най-често използваните оператори и опции в test:

Сравняване за цели числа	Функция
-gt (greater than)	По-голямо от
-lt (less than)	По-малко от
-ge (greater or equal)	По-голямо или равно
-le (less or equal)	По-малко или равно
-eq (equal)	Равно
-ne (not equal)	Неравно
Сравняване на стрингове	
-z	Проверява за празен стринг
=	Равни стрингове
!=	Неравни стрингове
Проверки за файлове	
-f	Файлът съществува
-s	Файлът не е празен
-r	Файлът може да се чете
-w	Във файлът може да се записва
-x	Файлът може да се изпълнява

Има много повече опции, но тук няма да ги описвам. Можете да прочетете за тях в test manual pages (man test).

Условието **if** проверява изходното състояние на команда. Ако то е нула командите в структурата се изпълняват.
Ако е нещо различно от нула командите в структурата не се изпълняват. Всяка if структура завършва с ключова дума
fi и всяка case структура завършва с esac. Контролните структури за условие:

if команда then команда fi	Втората командата се изпълнява ако изходното състояние на първата команда е 0.

if команда then
команда
else
команда
fi
	Ако изходното състояние е 0 се изпълнява командата в if, ако е някакво друго число се изпълнява командата в else.

if команда then команда	elif ви позволява да проверявате няколко ситуации в една if структура.

```
elif команда then
  команда
else
  команда
fi

case стринг in          case търси съвпадение на стринга с някои от шаблоните, ако няма такова
шаблон)                 се изпълняват командите по подразбиране (не е задължително да има такива.
  команда;;
*)
  команда по подразбиране;;
esac
```

команда && команда — логическо И (AND)

команда || команда — логическо ИЛИ (OR)

!команда — логическо НЕ (NOT)

По-долу са показани примери с горните условия:

```
#!/bin/bash

echo -n "Enter a string: "
read str1
echo -n "Enter a string: "
read str2
echo -n "Enter a number: "
read num1

if [ $str1 == "m000" ]; then  # ';' се използва за да може then да е на същия ред
    echo "str1 = m000"
elif [ $str1 == "m000" ] && [ $str2 == "m000" ]; then # логическо И, т.е. echo "str1 and str2 = m000" ще се
                        # изпълни ако и двете условия са true
      echo "str1 and str2 = m000"
    else
     echo "str1 and str2 != m000"
    fi

if [ -f "/etc/passwd" ]; then # пример с файлове
      cat /etc/passwd
    fi

if [ $num1 -eq 10 ]; then
      echo "num1 = 10"
elif [ $num1 -gt 100 ]; then
   echo "num1 > 100"
else
   echo "?!?"
fi
```

Изпълнението на скрипта:

```
$ ./primer12
Enter a string: m000
Enter a string: m000
Enter a number: 10
str1 = m000
root:x:0:0::/root:/bin/bash
```

```
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/log:
nobody:x:99:99:nobody:/:
sftp:x:1000:100:,,,:/home/sftp:/bin/bash
num1 = 10
$
```

Можете да пробвате скрипта като въвеждате други стойности. Един пример за **case**:

```
#!/bin/bash

echo -n "Enter an option (l, s or al): "
read opt

case $opt in
l)
  ls -l;;
s)
  ls -s;;
al)
  ls -al;;
*) # ако $opt не съвпада с никоя от горните опции
  ls;;
   esac
```

Изпълнението на скрипта:

```
$ ./primer13
Enter an option (l, s or al): l
total 964
drwxr-xr-x   33 sftp     users       4096 май 24 19:58 books
drwxr-xr-x    5 sftp     users       4096 юни 12 19:35 docs
drwxr-xr-x    7 sftp     users       4096 май  6 12:34 ircd
drwxr-xr-x    2 sftp     users       4096 май 25 23:22 movies
drwxr-xr-x    7 sftp     users       4096 май 17 11:56 phun
drwxr-xr-x   10 sftp     users       4096 юни  8 15:54 source
drwxr-xr-x    2 sftp     users       4096 юни  1 15:27 stuff
drwxr-xr-x    2 sftp     users       4096 юни 15 08:40 work
$
```

Това е всичко за условията. Можете да пробвате другите оператори и опции на test сами.

##

[--- **Цикли** ---]

Циклите се използват за повтаряне на команди. Контролните структури за цикли са while, until, for и for-in. while и until проверяват резултата на някаква команда докато for и for-in обхождат списък от стойности като присвояват всяка стойност на някаква променлива. Структурите за цикъл са показани по-долу:

```
while команда do          while изпълнява дадена команда докато връщаната стойност от
  команда                 първата команда е 0 (true).
done

until команда do          until изпълнява дадена команда докато връщаната стойност от
  команда                 първата команда е 1 (false).
done

for променлива in списък от стойности   На променливата се присвоява последователно всяка от
```

стойностите
```
do                  в списъка.
  команда
done
```

```
for променлива      for е предназначен за обръщане към аргументите на скрипта. На
 do                 променливата се присвоява последователно всеки от аргументите.
  команда
done
```

while изпълнява команди докато изходната стойност на първата команда е 0. Края на цикъла се указва с ключовата
дума done.

```
#!/bin/bash

m00=yes

while [ $m00 == "yes" ]; do
  echo -n "Enter a string: "
  read str1
  echo "You entered: $str1"
  echo -n "Do you want to continue? "
  read m00
done
```

Скриптът ще се изпълнява докато въвеждате yes, при всяка друга стойност изпълнението спира защото връщаната
стойност от [$m00 == "yes"] ще е 1 (false). Изпълнението на скрипта е показано тук:

```
$ ./primer14
Enter a string: asd
You entered: asd
Do you want to continue?  yes
Enter a string: asd1234123
You entered: asd1234123
Do you want to continue? no
$
```

Един пример с until:

```
#!/bin/bash

m00=yes

until [ $m00 == "no" ]; do
  echo -n "Enter a string: "
  read str1
  echo "You entered: $str1"
  echo -n "Do you want to continue? "
  read m00
done
```

Този скрипт извършва същата функция като предния, само че се изпълнява докато [$m00 == "no"] е false.

```
$ ./primer15
Enter a string: me0w
You entered: me0w
Do you want to continue? yes
Enter a string: zmpf
```

```
You entered: zmpf
Do you want to continue? no
$
```

for-in се използва за обхождане на списък от стойности като всяка от стойностите се присвоява последователно
на променлива.

```
#!/bin/bash

for cfile in ~/source/c/*.c # това е същото като for cfile in $( ls ~/source/c/*.c )
do
  echo $cfile
done
```

Изпълнението на скрипта:

```
$ ./primer16
/home/sftp/source/c/a.c
/home/sftp/source/c/tmp.c
/home/sftp/source/c/zmpf.c
$
```

for е същото като for-in, но обхожда аргументите на скрипта.

```
#!/bin/bash

for m00
do
  echo $m00
done
```

Тъй като аргументите на скрипта се съхраняват в специалната променлива $@, горният скрипт може да се направи и
с for-in.

```
#!/bin/bash

for m00 in $@
do
  echo $m00
done
```

Изпълнението на скрипта:

```
$ ./primer17 m00 me0w m33p zmpf lkmnp
m00
me0w
m33p
zmpf
lkmnp
$
```

Има още няколко команди, които често се използват в циклите. Това са командите true, false и break. Командата
true има много проста функция - изходната ѝ стойност е винаги 0. false е същата, само че нейната изходна стойност е 1. break се използва за излизане от цикъла.

```
#!/bin/bash

  declare -i var=0

  while true; do # тъй като никога не може условието да стане false цикълът е безкраен
    echo -n $var
    var=$var+1
    if [ $var -eq 10 ]; then # ако $var е 10 цикълът се прекратява
      break
    fi
  done

  echo
  echo bye
```

Можете да направите същия цикъл ако използвате until и false единствената разлика е в реда:

while true; do

Трябва да е

until false; do

##

[--- **Пренасочване и канализиране** ---]

Когато изпълнявате команда, тя изпраща данните на стандартния изход (конзолата). Но можете да пренасочите данните и
към файл или примерно към stderr. Също така данните, които въвеждате могат да се вземат от файл. Пренасочването на
изходните данни се извършва с знака > след командата.
Пример:

```
  $ echo m000 > file
  $ cat file
  m000
  $
```

Ако искате да добавите данни към съществуващ файл използвайте >>, защото > изтрива предишното съдържание на файла.

```
  #!/bin/bash

  nums="1 2 3"

  for num in $nums
  do
    echo $num
  done > nums
```

Горния скрипт ще пренасочи изходните данни в файла nums.

```
  $ ./nums
  $ cat nums
  1
  2
  3
  $
```

Канализирането е подобно на пренасочването, с тази разлика че изходните данни от една команда се предават като входни
на друга. Най-прост пример за канализиране е архивирането и дезархивирането с tar и gzip.

```
$ tar cvf - files/ | gzip -9 > files.tar.gz
```

или

```
$ gzip -cd files.tar.gz | tar xvf -
```

Един пример с bash скрипт.

```
#!/bin/bash

names="Ivan Koljo Asen Sasho Misho"

for name in $names
do
  echo $name
done | sort > names
```

Скриптът предава изходните данни от echo $name на командата sort и след това изходните данни от нея се пренасочват във
файла names.

```
$ ./names
$ cat names
Asen
Ivan
Koljo
Misho
Sasho
$
```

Както виждате имената са подредени по азбучен ред тъй като изходните данни се предават на командата sort.

##

[--- Структурата **trap** ---]

Друг вид структура е trap. trap изпълнява команда при някакво събитие. Тези събития се наричат сигнали. Тук е даден синтаксисът на командата trap.

```
$ trap 'команди' номер-на-сигнала
```

Един често използван сигнал е сигналът за прекъсване (когато потребителят натисне CTRL+C).

```
#!/bin/bash

trap 'echo "Bye"; exit 1' 2

echo "Enter a string: "
read m00
echo "nah"
```

Изпълнението на скрипта:

```
$ ./trap-ex
Enter a string:
^CBye
$
```

Както виждате след като натиснем CTRL+C се изпълнява echo "Bye" след това exit 1, с което програмата спира изпълнението си. Списък с различни номера на сигнали е показан тук:

```
0 Край на програмата      Програмата завършва
2 Терминално прекъсване   CTRL+C
9 kill            kill-ване на програмата, не може да се улови с trap
24              Спиране на програмата с CTRL+Z, не може да се улови
```

Можете да видите повече номера на сигнали за kill в kill manual pages.

##

[--- **Масиви** ---]

Масивът може да бъде инициализиран с var[xx]="aa", където xx е някакво число или с командата declare -a променлива.
За да се обърнем към стойността на някой от елементите на масива трябва да го заградим във фигурни скоби ('{' и '}').
Пример:

```
#!/bin/bash

array[1]=m00
array[34]=me0w
array[40]=123

echo "array[1] = ${array[1]}"
echo "array[34] = ${array[34]}"
echo "array[40] = ${array[40]}"
```

Друг начин за присвояване на стойности на елементите от масив е

```
array=( x y z ... )
```

След това x ще бъде array[0] (забележете първият елемент е 0 а не 1), y - array[1], z - array[2] и т.н.

Един пример, демонстриращ повече възможности на масивите:

```
#!/bin/bash

declare -a array # декларираме масив с declare

echo -n "Enter some numbers separated by space: "
read -a array # присвояваме въведените числа на елементите на масива (забележете опцията на read '-a')

elements=${#array[@]} # тук присвояваме на променливата elements броя елементи на масива

# ${array[@]} съдържа елементите на масива поотделно. Можете да го използвате за цикъл for-in
# например:
# for i in ${array[@]; do
#    echo $i
# done

# сега ще покажем елементите на масива с цикъл while
```

```
i=0

while [ $i -lt $elements ]; do
  echo ${array[$i]}
  let "i = $i + 1"
done

echo "bye"
```

Изпълнението на скрипта:

```
$ ./primer18
Enter some numbers separated by space: 1 13 14 88 19
1
13
14
88
19
bye
$
```

Можете да видите много повече за масивите (различни алгоритми за сортиране и др.) в Advanced Bash Scripting
(http://www.tldp.org/LDP/abs/html/arrays.html)

##

[--- **Функции** ---]

Както в повече програмни езици и в bash можете да използвате функции за да групирате няколко команди в една.
Декларирането на функция става така:

```
име_на_функция { команди }
```

или

```
име_на_функция() { команди }
```

За да извикате функция просто трябва да въведете нейното име.

```
#!/bin/bash

func() {
  echo "Hello"
  echo "This is function"
}

info() {
  echo "host uptime: $( uptime )"
  echo "date: $( date )"
}
func
info

echo "That's all folks :)"
```

Изпълнението на скрипта:

```
$ ./primer19
Hello
This is function
host uptime:   1:46pm  up 5 days, 14:39,  1 user,  load average: 0.08, 0.05, 0.04
date: сб юни 15 13:46:23 UTC 2002
That's all folks :)
  $
```

##

[--- **Цветове** ---]

И последната част от tutorial-а са цветовете в bash. За да изкараме цветен текст ще използваме командата printf.
Цвета се определя с e[X;XXm, където X-овете са числа. Списъка с цветовете е показан по-долу:

Черно	0;30	Тъмно сиво	1;30
Синьо	0;34	Светло синьо	1;34
Зелено	0;32	Светло зелено	1;32
Cyan	0;36	Light Cyan	1;36
Червено	0;31	Светло червено	1;31
Лилаво	0;35	Светло лилаво	1;35
Кафяво	0;33	Жълто	1;33
Светло сиво	0;37	Бяло	1;37

Изчиства екрана 2J (thanks zEAL)

Пример:

```
#!/bin/bash

printf "e[2J"
printf "e[0;34mbluen" # n е за нов ред
printf "e[0;32mgreenn"
printf "e[0;37mand back to default color ...n"
```

BASH - въведение в конзолата,терминал
от cleaver на Чет Юни 12, 2008 20:48

Конзолата е естественото средство за комуникация между човека и машината във всяка една Unix-базирана система. По-правилно е да се нарича "команден интерпретатор". С негова помощ, чрез писане на различни команди, ние управляваме компютъра. А той от своя страна говори на нас, като изписва резултатите от изпълнението на нашите команди било на екрана (в конзола), било във файл. Но за това малко по-късно. В Линукс има различни командни интерпретатори. Най-разпространеният от тях - BASH (който е и предмет на настоящата статия) в качеството си на команден интерпретатор е просто една програма, която интерпретира команди. Но освен това BASH е и мощен скриптов език, който предоставя почти неограничена гъвкавост при работа с тези команди. Както всеки друг език, така и BASH си има синтаксис, който трябва да се спазва, за да бъде това, което ние пишем, разбираемо и изпълнимо за компютъра. Така например в най-общия случай на всеки команден ред се пише по една команда, чието писане приключва с натискането на клавиша Enter и в същия този момент започва нейното изпълнение. Има обаче специални символи, които служат за управление работата на BASH и не могат да се използват направо в команди. Един от тези символи - ;, служи за разделяне на две последователни команди. Той позволява две и повече команди да се напишат на един ред, като между тях се поставя този символ. И когато след последната команда натиснем клавиша Enter, всички изброени команди се изпълняват поред. Друг специален символ - \, позволява една команда да се напише на два и повече реда. Когато този символ се постави в края на даден ред и след това се натисне Enter, това не води до изпълнение на команда, а просто поставя курсора на нов ред, където продължаваме да пишем същата команда. Когато сме готови с написването, можем да изпълним командата, като натиснем Enter без \ пред него.
Бих искал да обърна внимание на 3 много важни и основни за всеки шел символа.
Символът '~' (tilde) означава домашната директория. Така вместо '/home/myname' можем да напишем само '~', същото е.
Символът '.' (точка) означава текущата директория. С него винаги можем да я означим, без да изписваме целия път до нея.
Символът '..' (две точки) означава директорията, която съдържа текущата.
Всички файлове и директории, чиито имена започват с '.', са скрити и по подразбиране не се виждат.

Какво представляват командите?

В някои случаи това са функции, вградени в самия BASH (builtins), но много по-често това са програми, които се намират някъде по системата и просто се извикват от BASH при положение, че знае къде да ги търси. Почти всички команди в BASH имат сходен синтаксис, което улеснява тяхното научаване и използване. Всяка команда се състои от име, към нея може да се добавят опции и евентуално може да приема някакви аргументи. Някои команди допускат в синтаксиса си специални аргументи, които се наричат подкоманди. Какво представляват опциите? Има така наречените "къси опции" и "дълги опции". И двата вида се използват за добавяне на различни детайли към изпълнението на командата и позволяват да посочим конкретни специфични особености за нейното изпълнение. С опциите казваме на командата конкретни подробности, свързани с нейното изпълнение. Най-лесно е да го разберем така - всяка команда има своя функционалност и може да прави много различни неща. Именно чрез добавянето на опции към синтаксиса на командата ние посочваме кои неща да направи (и евентуално как) и кои не. Всяка команда си има специфични за нея (и точно определен брой) опции, но при всички команди синтаксисът им е един и същ. Късите опции обикновено представляват буква или къс низ, предхождани от тире, а дългите опции са низ (обикновено дума), предхождан от две последователни тирета. Аргументите на командите са нещо като променливи. В повечето случаи тяхната стойност зависи от това, което искаме да направим в момента. С тях обикновено се посочват нещата, които афектира (за които се отнася) нашата команда и какво точно касае тя. За да изясним всичко това, ще дадем просто пример с една тривиална команда за копиране на файлове и директории. Това е командата 'cp'.

```
cp -ri --verbose dir1 /destination/directory/
```

Тази команда в случая копира dir1 в директорията, написана след нея. В случая dir1 и /destination/directory/ се явяват аргументи на командата, защото показват за какво (в случая за кои директории) се отнася тя. Имаме и общо 3 опции - 2 къси и една дълга. Както виждаме в случая, възможно е две и повече къси опции да се въвеждат с едно тире, стига да няма интервал между тях. Тази част '-ri' може да се напише и така '-r -i', като всяка къса опция се въвежда със свое собствено тире. Дългата опция --verbose се въвежда с две тирета. Сега

по две думи за опциите. Опцията -r ще рече 'recursive'. Тя казва на командата 'cp' да копира цялото съдържание на dir1, включително нейните поддиректории, ако има такива. Опцията -i означава 'interactive'. Това означава, че командата ще ви задава въпроси, когато не знае какво да прави (например дали да презапише съществуващ файл) и вие ще имате възможност да отговорите (обикновено чрез натискане на 'y' или 'n' за 'да' или 'не'). Опцията --verbose казва на 'cp' да изписва подробно на изхода си (обикновено екрана) какво прави. Някои опции имат къси и дълги версии, стига някоя от тях да не се дублира с друга опция. Например опцията '--verbose' може да се зададе и като '-v'.

Пътища

Тук искам да спомена някои важни неща за работата с пътища в BASH. Чрез пътищата се указва точното местоположение на даден файл или директория във файловата система. В горния пример с 'cp' обърнете внимание на втория аргумент - /destination/directory/. Тази директория е описана с така наречения абсолютен път до нея. Абсолютен е защото тръгва от корена - /. Всеки път до файл или директория, който започва с корена на файловата система, е абсолютен. И независимо къде се намираме в момента по файловата система, чрез абсолютен път винаги може да се опише точното местонахождение на всяко едно нещо във файловата система, защото тръгваме от корена и после просто минаваме по тези клони, които водят до това, което ни трябва. Обаче този начин за описване на пътища в повечето случаи е много дълъг, защото винаги тръгваме от корена, а това невинаги е необходимо. Обърнете внимание на първия аргумент в примера - dir1. Той е описан с така наречения 'относителен път'. Нарича се относителен, защото използването му зависи от това къде се намираме в момента. Както виждаме, не започва с корена (пред него няма /), защото директорията, в която се намираме в момента, играе роля на корен. Например ако абсолютният път до 'dir1' е да кажем '/usr/source/dir1', то ние трябва да се намираме в директория '/usr/source', за да укажем пътя относително, както сме направили по-горе. Ако се намирахме в директория '/usr/proba/' например, горната команда би върнала грешка, защото директория 'dir1' не се намира в 'proba', а в съседен на нея клон - 'source'. С други думи, за да можем да окажем относително пътя до даден обект във файловата система, ние трябва да се намираме на такъв клон, от който имаме пряк достъп до обекта. В противен случай трябва да окажем пътя като абсолютен, тръгвайки от истинския корен. Защото само истинският корен има достъп (първоизточник е на) всички клонове.

Вход/изход. Пренасочване.

По подразбиране почти всички команди приемат вход от клавиатурата и пращат изхода си на екрана. Това са стандартните input и output устройства. Но така е само по подразбиране. Има специални оператори, с които входът и изходът могат да се управляват и насочват. Например с команда 'ls' можем да отпечатаме съдържанието на дадена директория. Само че вместо да го гледаме на екрана, както би било по подразбиране, ние можем да пренасочим изхода от командата и да го запишем във файл. Това ще направим посредством оператора '>'. Ето пример:

```
ls > list.file
```

Команда 'ls' с опция '-l' ни извежда списък с всички файлове в директорията заедно с техните атрибути. А команда 'grep' филтрира този списък (изхода на ls) и ни показва само тези редове, които съдържат '.jpg' (които ни интересуват).
Съществуват и други оператори за управление на входа и изхода в BASH, но тези двата са най-често използвани и най-популярни и останалите не са предмет на разглеждане в настоящия труд. Все пак е важно да запомните, че когато обстоятелствата предполагат, няколко оператора могат да се комбинират в една команда, което може да ни предостави изключителна гъвкавост при изпълнение на по-сложни задачи. Тук веднага ще дам пример за една такава малко по-сложна задача (комбинираща два оператора), тъй като съм забелязал, че някои автори избягват да дават подобни примери. Да предположим, че искаме да изведем съдържанието на някаква директория, като филтрираме само .jpg снимките, след което да го сортираме по азбучен ред наобратно и накрая да го запишем като текстов файл в архив с някакво име. Всичко това можем да направим накуп със следната команда:

```
ls | grep .jpg | sort -r | gzip > /home/myname/Desktop/lsss.gz
```

Какво се случва? 'ls' извежда съдържанието на текущата директория, след което го подава на 'grep', която подбира само тези редове, които завършват на '.jpg' и ги изпраща на 'sort' да ги сортира наобратно. Накрая 'sort' подава сортираните редове (завършващи на '.jpg') на 'gzip', която ги компресира и ги записва в архив с име lsss.gz на нашия десктоп.

По-важни команди:
След като изяснихме някои основни принципи на BASH, нека да разгледаме и някои от най-често използваните команди и тяхното предназначение.

1. Команди за извличане на информация.

pwd (=Print Working Directory) - показва абсолютния път до текущата директория.
hostname - показва името на локалния хост (компютъра, чийто физически терминал използваме).
whoami - изписва името на текущия потребител (този, който въвежда командата).
date - показва и/или променя текущия час и дата.
time - в комбинация с някоя друга команда след нея показва за колко време се изпълнява дадената команда.
who - показва кои потребители са включени в момента.
finger [user_name] - показва системна информация за даден потребител.
last - извежда хронологичен списък с потребители, които последно са се включвали в системата, както и точния час и дата на включването им.
history - показва списък с последните изпълнени от BASH команди.
uptime - показва времето, изминало от последния рестарт на системата до момента.
ps - извежда списък с процесите, управлявани от текущия терминал. Изпълнена с опция -e, показва всички процеси.
top - показва динамичен списък с активните процеси (заедно с информация за всеки от тях), както и полезна статистика и информация за системното натоварване (процесор, памет).
uname - показва името на операционната система. С опция -a, показва и версията на ядрото, името на хоста, архитектурата и пр.
free - показва информация за паметта.
df -h - показва заетото и свободното място на всяка монтирана файлова система.
du -h /dir - изчислява размера на директория, мястото, което всички поддиректории и файлове в нея заемат.

Базови команди.

cat - извежда съдържанието на един или повече файлове.
echo - извежда текст, който й е бил подаден като аргумент.
man - извежда подробна помощна информация за командата, която й е подадена като аргумент.
ls - извежда списък със съдържанието на текущата директория. С опция -l извежда подробен списък, с опция -a показва и скритите файлове и директории.
cd [directory] (=Change Directory) - смяна на текущата директория. Използвана без аргумент, командата връща в домашната директория.
cp [file1 file2 ...] [destination] - копира файлове. С опция -r (=recursive) копира цели директории със съдържанието им.
mv - преименува файлове и директории или ги мести от едно място на друго.
rm - изтрива един или повече файлове. С опция -r (=recursive) изтрива цели директории с тяхното съдържание.
less - с нея можете да преглеждате текстови файлове, без да ги редактирате. Удобна е за четене на големи файлове.
find / -name "filename" - търси за файлове с име "filename" в дадена директория (в случая /) и нейните поддиректории.
touch - създава празен файл, чието име се подава като аргумент на командата.
kill PID - спира процес по зададен номер на процеса. С опция -9 процесът бива убит незабавно.
killall - спира процес, чието име се подава като аргумент на командата. Важи за всички процеси с това име.
useradd - създава нов потребителски профил, чието име се подава като аргумент на командата.
userdel - изтрива потребителски профил, чието име се подава като аргумент на командата.
passwd - създава/променя паролата на потребителски профил. Всеки профил (с изключение на root) може да променя само своята парола. root може да променя паролата на всеки потребител.
chmod - променя права на файлове и директории.

Bash Хитринки http://wiki.linuxfan.org/index.php/Bash

Изрязване на името на файл от пълния път

```
$ FULLPATH='/usr/local/etc/rc.d/runme.sh'
$ basename ${FULLPATH}

runme.sh
```

Изрязване само на директориите от пълния път

```
$ FULLPATH='/usr/local/etc/rc.d/runme.sh'
$ echo ${FULLPATH%/*}

/usr/local/etc/rc.d
```

Скриптове

Скрипт за предоставяне достъп до машини зад NAT

```
#!/bin/bash
# opengate.ssh script
# redirect ports for NAT-ed hosts
/bin/ping -i 30 linuxfan.org 1> /dev/null 2> /dev/null &
while (true) do
  /usr/bin/ssh -n -N -R 2222:127.0.0.1:22 -R 2223:127.0.0.1:10001 remote_shell@linuxfan.org
  logger $0 disconneted!
  sleep 30
done
```

Скрипт за фонетична кирилизация на X (и за махане на Caps Lock)

```
#!/bin/bash
    # a script to setup your language in x window
    # setup the locale
    unset lc_all
    lang=bg_bg
    language=bg:en_uk
    export lang language
    # setup the keyboard
    setxkbmap -types complete -compat 'complete+leds(scroll)' -geometry 'pc(pc102)' \
      -symbols \
    'en_us(pc101)+bg(phonetic_enhanced)+group(alt_shift_toggle)+group(rwin_switch)+level3(menu_switch)'
    # nls for old motif applications
    if test -d /usr/x11r6/lib/x11/nls; then
       xnlspath=/usr/x11r6/lib/x11/nls
       export xnlspath
    fi
    # add custom user fonts path
    if [ -f ${home}/.fonts/fonts.dir ]; then
       xset fp+ ${home}/.fonts
    fi
    # add system-wide fonts path for cyrfonts (this is not for debian)
    if [ -f /opt/cyrfonts/fonts.dir ]; then
       xset fp+ /opt/cyrfonts
    fi
    # read the user resources database
    if test -f ~/.xresources; then
       xrdb -merge ~/.xresources
    fi
    # disable caps-lock. evil key :-)
    xmodmap -e "remove Lock = Caps_Lock"
```

Скрипт за смяна на default gateway

```
#!/bin/sh
# stupid script to change default gateway
# author t0d0r at linuxfan dot org
# vim:ts=4:sw=4:

primary_gateway='217.79.66.2'
secondary_gateway='10.210.0.1'

ping_cmd='/sbin/ping'

change_to() {
      /sbin/route delete default
      /sbin/route add default $1
}

current_gateway=`/usr/bin/netstat -rn | /usr/bin/grep default | /usr/bin/awk '{print $2}'`

${ping_cmd} -c 3 ${current_gateway} 2>&1 > /dev/null && is_ok=true
if [ "x_${current_gateway}" = "x_${primary_gateway}" ];  then
   if [ "x${is_ok}" = "xtrue" ]; then
      exit;
   else
${ping_cmd} -c 3 ${secondary_gateway} 2>&1 > /dev/null \
   && change_to ${secondary_gateway}
exit;
fi
else
   ${ping_cmd} -c 3 ${primary_gateway} 2>&1 > /dev/null \
      && change_to ${primary_gateway}
   exit;
fi
```

Скрипт за проверка на сигнал до клиент

```
#!/usr/local/bin/bash
#Script for test connection from gateway to client
#Author: Peter Petrov aka peterpet
OS_TYPE=`uname`
PW=( 64 1024 1400 19000 25100 )
ping_l=`which ping`
if [ "x$1" = "x" ]; then
echo "Използване: ./vrazka <IP adress>"
exit 1
fi
if [ "$OS_TYPE" = "Linux" ]
then
echo "Работим на Linux"
echo
echo ".....ЗАПОЧВАМЕ ТЕСТ НА ВРЪЗКАТА ЗА ЗАГУБИ ...."
echo
echo

    for i in ${PW[@]}
    do
    iz=`${ping_l} -c 10 -s $i $1 | grep loss | awk '{print $6}'`
    echo ""
        if [ "$iz" = "100%" ];
        then
        echo "КЛИЕНТА ИМА FIREWALL"
        echo "или му е изключен кабела"
        exit
        fi
    echo "Натоварване на "$1" с пакет "$i" е: "$iz" "
done
fi

if [ "$OS_TYPE" = "FreeBSD" ]
then

echo "Работим на FreeBSD"
INTERFACE=`route get $1 | grep interface | awk '{print $2}'`
echo "$INTERFACE"
sudo /usr/local/sbin/arping -c 3 -i "$INTERFACE" "$1"

echo
echo ".....ЗАПОЧВАМЕ ТЕСТ НА ВРЪЗКАТА ЗА ЗАГУБИ ...."
echo
echo

     for i in ${PW[@]}
    do
    iz=`sudo ${ping_l} -c 10 -s $i $1 | grep loss | awk '{print $7}'`
    echo ""
        if [ "$iz" = "100%" ];
        then
        echo "КЛИЕНТА ИМА FIREWALL"
        echo "или му е изключен кабела"
        exit
        fi
    echo "Натоварване на "$1" с пакет "$i" е: "$iz" "
done
```

```
fi
echo ""
echo ""
echo "############################ КРАЙ НА ПРОВЕРКАТА ########################"
exit
```

Скрипт за differential backup

```
#!/bin/bash
# backup script by todor.dragnev at gmail.com
# useful for daily backup of AWS instances
prefix=/vol/backup/instances/`hostname`
exclude_file="$prefix/rsync.exclude"
compare_dest=${prefix}/`date +"%Y-%m-01"`/
source=/
target=$prefix/`date +"%Y-%m-%d"`/
# do differential backup
if [ -d $compare_dest ]; then
rsync -ab --compare-dest=${compare_dest} \
  --exclude-from=${exclude_file} \
  ${source} ${target}
else
# do full
  rsync -ab \
    --exclude-from=${exclude_file} \
    ${source} ${compare_dest}
fi
```

http://rosoft.org/node/57 02/25/2010 – 19:05 добавил: rrr

Команден интерпретатор в Линукс. Основни команди за работа в конзолен режим.

1. Основни понятия при изпълнението на команди в Линукс

конзола - това е исторически термин зад който стои дългогодишната история на компютъра въобще. В началото под конзола се е разбирало терминала на който е стоял администратора на машината и на който са излизали системните съобщения. Най-общо казано конзолата е текстово изходно устройство, което прихваща системните съобщения идващи от ядрото на операционната система или от програмата за системни съобщения. В големите компютри, конзолата най-често е била свързана със серийна връзка, работеща по стандарта RS-232 (серийния порт на компютъра). В съвременните персонални компютри под конзола се разбира монитора на компютъра. В Линукс съществува и понятието виртуална конзола, като това са шесте терминални прозореца, които може да се превключват с Alt+F1 до F6. Важно е да се отбележи, че системните съобщения излизат на първа виртуална конзола даже и никой да е логнат в нея. Типичен пример за това са системните съобщения на ядрото, които се появяват при стартирането на системата.

терминал - устройство (най-често хардуерно), което позволява комуникацията с компютъра. По принцип терминала е комбинация от монитор и клавиатура. Това не е така, обаче когато се говори за отдалечен терминал. Отдалечения терминал представлява специална програма съставена от клиент и сървър, чрез която потребителя може да се свързва отдалечено към компютър в който има валидно потребителско име и парола. Като за начало трябва да знаете, че най-често използваните програми са telnet и ssh.

- команден интерпретатор (шел) - програмата с помощта, на която системата комуникира с потребителя. Тя чете въведените от терминала редове и изпълнява различни операции в зависимост от това което е въведено. След това, шела се опитва да преобразува въвежданите конструкции в инструкции, които ядрото е в състояние да разбере.

Всеки потребител при логическото си включване в системата стартира свое копие на шел в паметта. Това се прави, за да може той да работи без да пречи на останалите потребители на системата.

Шела възприема всичко до първия интервал като команда, а всичко останало, като аргументи на тази команда, като аргументите също се разделят с интервали.

Всички (или поне повечето) Линукс системи имат повече от един валиден шел. Те са описани във файла /etc/shells. Той има следния прост вид:

```
/bin/bash

/bin/tcsh

/bin/csh

/bin/ash

/bin/ksh

/bin/zsh

/bin/rbash
```

Линукс система може да има повече инсталирани командни интерпретатори, но те са недостъпни за използване ако не са описани в този файл. Всеки потребител може да промени своя шел чрез командата chsh. Без параметри командата влиза в интерактивен режим на работа, а чрез опцията -s, този режим се изключва. Формата на командата е следния:

```
#chsh -s login_shell user
```

или само

#chsh

В някои дистрибуции е възможно проверка на валидните шелове чрез ключа -l към командата chsh, но Slackware не поддържа този ключ.

Въпреки, че съществуват множество командни интерпретатори, то не всички се използват масово. Най-често използваните командни интерпретатори са:

· bash – bash е команден интерпретатор съвместим с шела sh. Той може да изпълнява команди, както от ред, така и от файл. Bash притежава мощен скриптов език, чрез който може да се пишат програми, изпълнявани директно от командния интерпретатор. Освен това bash притежава и функции, които са присъщи на други командни интерпретатори като ksh и csh.

· csh – това е шел, който обединява типичните функции за един команден интерпретатор като автоматично завършване на файлов имена при натискането на Tab, управление на задачите и поддръжка на история на командите със синтаксиса на C.

· tcsh – подобрена версия на csh, включваща допълнително корекция на синтаксиса и др., като запазва пълна съвместимост с оригиналния csh. Повечето дистрибуции създават символична връзка csh, сочеща към tsch.

· ksh – KornShell е команден интерпретатор, поддържащ собствен език за програмиране и позволява изпълнение на команда както от команден ред, така и от файл.

· zsh – подобрена версия на ksh. Притежва множество предимства пред него, като функции, вградена синтактична проверка. За разлика от tcsh и csh, то zsh не е напълно съвместим с ksh.

Повечето от изброените командни интерпретатори (bash, ksh, zsh) създават символична връзка, която започва с r и завършва с името на шела (rbash, rksh, rzsh), която указва на командния интерпретатор да превключи в защитен режим. В този режим потребителя е силно ограничен, като ограниченията зависят от конкретния шел. Описанието на restricted шеловете в /etc/shells не се препоръчва, защото след преминаване към такъв шел, потребителя не може да се върне към обичайния за него команден интерпретатор.

Най-използвания от гореизброените командни интерпретатори е bash. Той е и шела по-подразбиране в Linux. Bash е абревиатура от Bourne Again Shell, създаден от Free Software Foundation на базата на Bourne shell от UNIX™.

2. Променливи на обкръжението

Всяка обвивка (обкръжение) има свои променливи на обкръжението. Техните стойности се установяват от командния интерпретатор при стартирането му и имат различни стойности за различните потребители. Ето ви още една причина за всеки потребител да се стартира отделно копие на интерпретатора. Стойностите на тези променливи могат да се променят в процеса на работа.

Например ако не ви харесва знака $ за промпт достатъчно е да смените променливата на обкръжението PS1.

Променливите на обкръжението може да се променят чрез командата:

$export PS1=%

%

само за текущата сесия или чрез промяна на файла .bash_profile или само .profile, като в този случай промяната става активна след като потребителя се логне и е постоянна. Slackware използва файла .bash_profile за въвеждане на команди или настройка на bash. Този файл се зарежда винаги когато потребителя влезе в системата. По-важните опции на bash са:

· -r, --restricted – указва на шела да влезе в ограничен режим. В този режим на потребителя са забранени следните операции: промяната на директории с командата cd; промяна на променливите на обкръжението SHELL, PATH, ENV и BASH_ENV; изпълнението на команди съдържащи знака /; задаването на / като аргумент на вградената команда .; задаване на файл съдържащ / като аргумент на вградената команда hash; импортиране на дефиниции на функции при стартиране; пренасочване на изхода с >, >|, <>, >& и >>; изпълнението на командата exec; изключване на режима на ограничения чрез командите set +r или set +o restricted. Освен по този начин режима на ограничения може да се включи и чрез изпълнението на rbash.

· -rcfile file – изпълнява командите от файла, вместо от стандартния .bashrc.

Всички командни интерпретатори има множество променливи на обкръжението. Тези променливи описват различни характеристики на системата. Те се четат от останалите програми, които ги използват за настройка на вътрешните си параметри, като език на интерфейса, локализация, тип на системата, версия на операционната система, текуща директория и др. Bash притежава множество променливи на обкръжението, по-важните от които са:

· BASH – указва пълното име на файла използвано за извикването на това копие на bash.

· BASH_VESRION – показва версията на командния интерпретатор.

· GROUPS – показва списък с групите на които е член потребителя.

· HOSTNAME – показва името на компютъра.

· OSTYPE – показва на каква операционна система се изпълнява копието на bash

· PWD – показва текущата директория

· RANDOM – при всяко извикване този параметър се променя от 1 до 32767 по случаен признак.

· UID – показва уникалния идентификационен номер на потребителя, който е стартирал копието на bash.

· HISTFILE – дефинира и показва файла в който се съхраняват изпълнените команди. По подразбиране това е ~/.bash_history.

· HISTSIZE – дефинира и показва максималното количество реда, които може да се съдържат във файла с историята на командите.

· HOME – дефинира и показва домашната директория на потребителя.

· LANG – дефинира локала на операционната система за всички категории. Отделните категории може да са различни от този параметър.

· LC_MESSAGES – дефинира езика на който да се показват различните съобщения.

· PATH – показва и дефинира пътя в който се търсят командите.

Четенето на тези променливи може да стане с командата echo:

```
$echo $HOME

/home/student

$
```

Записа на нова стойност става чрез командата export:

```
$export HOME=/home/ftp

$echo $HOME

/home/ftp

$
```

Важна особеност е това, че при четене от променливата е необходимо пред нея да се сложи знак за долар, докато при нейна промяна това не е необходимо.

Освен променливите на обкръжението, които се дефинират автоматично от командния интерпретатор, почти всички програми дефинира собствени променливи на обкръжението.

3. Скриптов език на bash

Bash притежава мощен език за програмиране, който позволява писането на скриптове за администриране на системата или за други дейности, които могат да се извършават автоматизирано. Bash скрипта може да представлява или поредица от команди поставени във файл, които се изпълняват последователно, или скрипт написан на езика на bash в който се съчетават външни команди, променливи, цикли, условия и вътрешни bash команди. Пример за прост bash скрипт е следната последователност от команди:

```
$cat > simple_script

echo "Building file...."

ls -l /dev > dev_list

echo "Number of files:"
```

```
wc -l dev_list

rm dev_list

echo "Exiting"

Ctrl+D

$chmod +x simple_script

$./simple_scipt

Building file....

Number of files:

2633 dev_list

Exiting

$
```

Тази проста поредица от команди са обединени в един текстов файл наречен шел скрипт. Този файл трябва да има права за изпълнение, за да може командния интерпретатор да го изпълни. Повечето шел скриптове обаче не са толкова прости. Примери за сложни скриптове са командите installpkg, pkgtool и т.н, които служат за добавяне, премахване или обновяване на програмите под Slackware. Чрез използването на командите kdialog и dialog на шел скриптовете може да се направи и графичен интерфейс съответно за KDE и за текстова конзола. Пример за такъв скрипт е pkgtool и netconfig.

Писането на шел скриптове е един от трудните аспекти на Линукс, но тяхното разбиране би довело до пълен контрол върху системата чрез прост набор от предварително написани скриптове.

4. Команди за работа с потребители и групи

- useradd - добавяне на потребител;

- userdel - изтриване на потребител;

- usermod - промяна на информацията за потребител;

- passwd - промяна на паролата за потребител. Паролите се пазят в кодиран вид в специален файл /etc/passwd или /etc/shadow.

- groupadd - добавяне на група;

- groupdel - изтриване на група;

- groupmod - промяна на информацията за група;

- groups - дава информация към коя група принадлежи дадения потребител. Информацията за групите се пази във файла /etc/group

- su - превключване към друг потребител. Командата su без параметър превключва към потребителя root. Излизане от този режим става с командата exit

5. Команди за работа с файлове и директории

- ls (dir) - дава списък с файловете в дадена директория. Параметри към командата: -l, -S и други. Заповече информация: man ls

- cd - сменя текущата директория

- mkdir - създава директория

- cat - извежда съдържанието на текстов файл

- head - извежда първите редове на текстов файл

- tail - извежда последните редове на текстов файл

- vi - прост текстов редактор. За режим на въвеждане се използва клавиша "i". За изход от редактора без запис на файла се използват последователно: Esc, :, q, Enter , за изход със запис - Esc, :, wq, Enter

- cp - копира файлове

6. Команди за промяна на правата за достъп

- chmod - задава права за достъп до даден файл или директория

- chown - задава принадлежност на файла или директорията

- chkgrp - задава група за принадлежност на файла или директорията

7. Други команди:

- echo - извежда текст на екрана. С помощта на символа ">" изходът на тази команда може да се пренасочи към файл. Така може да се създаде прост текстов файл. Например: echo Днес времето е хубаво > vreme.txt

- man - средство за получаване на справочна информация. Пример: man ls - дава информация за командата ls.

- df - дава информация за капацитета на монтираните устройства (root директорията, както и всички монтирани устройства не принадлежащи на Линукс, напр: /mnt/win_c - дял C: на Windows)

- free - дава информация за RAM паметта - обща, заета и свободна памет

- wall - изпраща съобщение до всички потребители в системата. Края на съобщението се задава с CTRL+D

- clear - изчиства екрана

- date - извежда текущата дата и час

- uptime - показва колко дни е работила системата от последния рестарт, броя потребители в момента, както и

средното натоварване на системата за последните 1, 5 и 15 минути

- w - извежда информация за действията на потребителите в системата

- ps- извежда справка за всички процеси в паметта

- mc- стартира Midnight Commander (ако е инсталиран в системата) - файлов мениджър, подобен на Norton Commander

- telnet, ssh - осъществява връзка с отдалечен компютър

- netstat - дава информация за активните в момента мрежови връзки

Писане на скриптове за BASH
шел : версия 1.2

Статията е взета от http://linux-bg.exco.net/

Тази статия е преведена с разрешението на автора и X_console.
Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/

Както всички шелове, който може да намерите за Linux BASH (Bourne Again SHell) е не само отличен команден интерпретатор но и език за писане на криптове. Шел(Shell) скриптовете ви позволяват максимално да използвате възможностите на шел интерпретатора и да автоматизирате множество задачи. Много от програмите, който може да намерите за Linux в последно време са шел скриптове. Ако искате да разберете как те работят или как може да ги редактирате е важно да рабирате синтаксиса и семантиката на BASH шела. В допълнение познаването на bash езика ви позволява да напишете ваши собствени програми, който да изпълняват точно това което искате.

Програмиране или писане на скрипове?
Хората който до сега не са се занимавали с програмиране обикновено не разбират разликата между програмен и скрипт език. Програмните езици обикновено са по-мощни и по-бързи от скрипт езиците. Примери за програмни езици са C, C++, и Java. Програмите който се пишат на тези езици обикновено започват от изходен код (source code) - текст който съдържа инструкции за това как окончателната програма трябва да работи след което се компилират до изпълним файл. Тези изпълними файлове не могат лесно да бъдат адаптирани за различни операционни системи (ОС). Например ако сте написали програма на C за Linux изпълнимият файл няма да тръгне под Windows 98. За да можете да я използвате тази програма се налага да прекомпилирате изходния код под Windows 98. Скриптовете (програмите писани на скрипт езици) също започват от изходен, но не се компилират в изпълними файлове. При тях се използва интерпретатор който чете инструкциите от изходния код и ги изпълнява. За жалост, поради това че интерпретатора трябва да прочете всяка команда преди да я изпълни, интерпретираните програми вървят като цяло по-бавно спрямо компилираните. Основното предимство на скриптовете се е, че лесно могат да бъдат пренаписани за други ОС стига да има интерпретатор за тази ОС. bash е скрипт език. Той е идеален за малки програми. Други скрип езици са Perl, Lisp, и Tcl.

Какво искате да знаете?

Писането на собствени шел скриптове изисква от вас да знаете най-основните команди на Linux. Трябва да знаете например как да копирате, местите или създавате нови файлове. Едно от нещата което е задължително да знаете е как да работите с текстов редактор. За Linux има множество текстови редактори най-разпространените от който са vi, emacs, pico, mcedit.

Внимание!!!
Не се упражнявайте в писане на скриптове като root потребител! Не се знае какво може да се случи! Аз няма да бъда отговорен ако вие по невнимание повредите вяшата система. Имайте това в предвид! За упражненията използвайте нормален потребител без права на root.

Вашият първи BASH скрипт
Първият скрипт който ще напишете е класическата "Hello World" програма. Тази програма изпечатва единствено думите "Hello World" на екрана. Отворете любимия си текстов редактор и напишете:

```
#!/bin/bash
echo "Hello World"
```

Първият ред от програмата казва че ще използваме bash интерпретатора за да подкараме програмата. В този случай bash се намира в /bin директорията. Ако bash се намира в друга директория на вашата система тогава направете необходимите премени в първия ред. Изричното споменаване на това кой интерпретататор ще изпълнява скрипта е много важно, тъи като той казва на Linux какви инструкции могат да бъдат използвани в скрипта. Следващото нещо което трябва да направите е да запишете файла под името hello.sh. Остава само да направите файла изпълним. За целта пишете:

```
xconsole$ chmod 700 ./hello.sh
```

Прочетете упътването за използване на chmod командата ако не знаете как да променяте правата на даден файл. След като сте направили програмата изпълнима я стартирайте като напишете:

```
xconsole$ ./hello.sh
```

Резултатът от която ще бъде следният надпис на екрана

Hello World

Това е! Запомнете последователноста от действия - писане на кода, записване на файла с кода, и променянето на файла в изпълним с командата chmod.

Команди, Команди, Команди
Какво точно направи вашата първа програма? Тя изпечата думите "Hello World" на екрана. Но как програмата го направи това? С помоща на команди. Единственият ред с команди беше echo "Hello World". И коя е командата? echo. Командата echo изпечатва всичко на екрана, което е получила като свой аргумент.

Аргумент е всичко което е написано след името на командата. В нашият случай това е "Hello World". Когато напишете ls /home/root, командата е ls а нейният аргимент е /home/root. Какво означава всичко това? Това означава че ако имате програма или команда, която изпечатва аргументите си на екрана то може да я използвате вместо echo. Нека да предположим че имаме такава команда която се казва foo. Тази комада ще изпечата своите аргументи на екрана. Тогава нашият скрипт може да изглежда така:

```
#!/bin/bash
foo "Hello World"
```

Запишете го и го направете изпълним с chmod след което го стартирайте:

```
xconsole$ ./hello
Hello World
```

Точно същият резултат. Всичко което правихте до сега е да използвате echo командата във вашия шел скрипт. Друга команда за изпечатване е printf. Командата printf позволява повече контрол при изпечатване на информацията, особено ако сте запознати с програмния език C. Фактически същият резултат от нашият скрипт можем да постигнем просто ако напишем в командния ред:

```
xconsole$ echo "Hello World"
Hello World
```

Както виждате може да използвате Linux команди при писането на шел скриптове. Вашият bash шел скрипт е колекция от различни програми, специално написани заедно за да изпълнят конкретна задача.

Малко по-лесни програми
Сега ще напишем програма с която да преместим всички файлове от дадена директория в нова директория, след което ще изтрием новата директория заедно със нейното съдържание и ще я създадем отново. Това може да бъде направено със следната последователност от команди:

```
xconsole$ mkdir trash
xconsole$ mv * trash
xconsole$ rm -rf trash
xconsole$ mkdir trash
```

Вместо да пишем последователно тези команди можем да ги запишем във файл:

```
#!/bin/bash
mkdir trash
mv * trash
rm -rf trash
```

```
mkdir trash
echo "Deleted all files!"
```

Запишете файла под името clean.sh. След като стартирате clean.sh той ще премести всички файлове в директория trash, след което ще изтрие директорията заедно със съдържанието и ще я създаде отново. Дори ще изпечата съобщение на екрана, когато свърши със тези действия. Този пример показва и как да автоматизирате многократното писане на последователности от команди.

Коментари във скрипта
Коментарите ви помагат да направите вашата програма по-лесна за разбиране. Те не променят нищо в изпълнението на самата програма . Те се използват единствено за да може вие да ги четете. Всички коментари в bash започват със символа: "#", с изключение на първия ред (#!/bin/bash). Първият ред не е команда. Всички редове след първия който започват със "#" са коментари. Вижте кода на следния скрипт:

```
#!/bin/bash
# tazi programa broi chislata ot 1 do 10:
for i in 1 2 3 4 5 6 7 8 9 10; do
   echo $i
done
```

Дори и да не знаете bash, вие веднага може да се ориентирате какво прави скрипта, след като прочетете коментара. Добре е при писане на скриптове да използвате коментари. Ще откриете, че ако ви се наложи да промените някоя програма която сте писали преди време, коментарите ще ви бъдат от голяма полза.

Променливи
Променливите, най-общо казано, са "кутии" който съхраняват информация. Вие може да използвате променливи за много неща. Те ви помагат да съхранявате информацията която е въвел потребителя, аргументи или числова информация. Погледнете този код:

```
#!/bin/bash
x=12
echo "Stoinosta na promenliwata x e $x"
```

Всичко което се случва е да присвоим на променливата x стойност 12 и да я отпечатаме тази стойност с комадата echo. echo "Stoinosta na promenliwata x e $x" изпечатва текущата стойност на x. Когато давате стойност на дадена променлива не трябва да има шпации между нея и "=". Ето какъв е синтаксиса:

име_на_променлива=стойност

Стойноста на променливата можем да получим като поставим символа "$" пред нея. В конкретния случай за да изпечатаме стойноста на& x пишем echo $x.

Има два типа променливи - локални променливи и променливи на обкръжението(environment). Променливите на обкръжението се задават от системата и информация за тях може да се получи като се използва env командата. Например:

xconsole$ echo $SHELL

Ще отпечата

/bin/bash

Което е името на шела който използваме в момента. Променливите на обкръжението са дефинирани в /etc/profile и ~/.bash_profile. С echo командата можете лесно да проверите текущата стойност на дадена променлива, биля тя от обкръжението или локална. Ако все още се чудите защо са ни нужни променливи следната програма е добър пример който илюстрира тяхното използване:

```
#!/bin/bash
echo "Stojnosta na x e 12."
```

```
echo "Az imam 12 moliwa."
echo "Toj mi kaza che stojnosta na x e 12."
echo "Az sym na 12 godini."
echo "Kak taka stojnosta na x e 12?"
```

Да предположим че в един момент решите да промените стойноста на x от 12 на 8. Какво трябва да направите? Трябва да премените навсяскъде в кода 12 с 8. Да но... има редове в който 12 не е стойноста на x. Трябва ли да променяме и тези редове? Не защото те не са свързани с x. Не е ли объркващо? По надолу е същия пример само че се използват променливи:

```
#!/bin/bash
x=12     # stoinosta na promenliwata x e 12 e x
echo "Stoinosta na  x e $x."
echo "Az imam 12 moliwa."
echo "Toj mi kaza che stojnosta na x e $x."
echo "Az sym na 12 godini."
echo "Kak taka stojnosta na x e $x?"
```

В този пример $x ще изпечата текущата стойност на x, която е 12. По този начин ако искате да промените стойноста на x от 12 на 8 е необходимо единствено да замените реда в който пише x=12 с x=8. Другите редовете няма да бъдат променени. Променливити имат и дуги полезни свойства както ще се убедите сами в последствие.

Условни оператори
Условните оператори ви позволяват вашата програма да "взема решения" и я правят по-компактна. Което е по-важно с тях може да проверявате за грешки. Всички примери до сега започваха изпълнението си от първия ред до последния без никакви проверки. За пример:

```
#!/bin/bash
cp /etc/foo .
echo "Done."
```

Тази малка шел програма копира файлът /etc/foo в текущата директория и изпечатва "Done" на екрана. Тази програма ще работи само при едно условие. Трябва да има файл /etc/foo. В противен случай ще се получи следния резултат:

```
xconsole$ ./bar.sh
cp: /etc/foo: No such file or directory
Done.
```

Както виждате имаме проблем. Не всеки който стартира вашата програма има файл /etc/foo на системата си. Ще бъде по-добре, ако вашата програма проверява дали файла /etc/foo съществува и ако това е така да продължи с копирането, в противен случай да спре изпълнението. С "псевдо" код това изглежда така:

```
if /etc/code exists, then
   copy /etc/code to the current directory
   print "Done." to the screen.
otherwise,
   print "This file does not exist." to the screen
   exit
```

Може ли това да бъде направено с bash? Разбира се! В bash условните оператори са: if, while, until, for, и case. Всеки оператор започва с ключова дума и завършва с ключова дума. Например if оператора започва с ключовата дума if, и завършва с fi. Условните оператори не са програми във вашата система. Те са вградени свойства на bash.

if ... else ... elif ... fi
Е един от най-често използваните условни оператори. Той дава възможност на програма да вземе решения от рода на "направи това ако(if) това условие е изпълнено, или(else) прави нещо друго". За да използвате

ефективно условния оператор if трябва да използвате командата test. test проверява за съществуване на файл, права, подобия или разлики. Ето програмата bar.sh:

```
#!/bin/bash
if test -f /etc/foo
then
   # file exists, so copy and print a message.
   cp /etc/foo .
   echo "Done."
else
   # file does NOT exist, so we print a message and exit.
   echo "This file does not exist."
   exit
fi
```

Забележете че редовете след then и else са малко по-навътре. Това не е задължително, но се прави с цел програмата да бъде по-лесна за четене. Сега стартирайте програмата. Ако имате файл /etc/foo, тогава програмата ще го копира в текущата директория, в противен случай ще върне съобщение за грешка. Опцията -f проверява дали това е обикновен файл. Ето списък с опциите на командата test:

-d проверява дали файлът е директория
-e проверява дали файлът съществува
-f проверява дали файлът е обикновен файл
-g проверява дали файлът има SGID права
-r проверява дали файлът може да се чете
-s проверява дали файлът разнерът на файла не е 0
-u проверява дали файлът има SUID права
-w проверява дали върху файлът може да се пише
-x проверява дали файлът е изпълним

else се използва ако искате вашата програма да направи нещо друго, ако първото условие не е изпълнено. Има и ключова дума elif, която може да бъде използвана вместо да пишете друг if вътре в първия if. elif идва от английското "else if". Използва се когато първото условие не е изпълнено и искате да проверите за друго условие.

Ако не се чувствате комфортно с if и test синтаксиса, който е :

```
if test -f /etc/foo
then
```

тогава може да използвате следния вариант:

```
if [ -f /etc/foo ]; then
```

Квадратните скоби формират test командата. Ако имате опит в програмирането на C този синтакс може да ви се стори по-удобен. Забележете, че трябва да има разстояние след отварящата квадратна скоба и преди затварящата. Точката и запетаята: ";" казва на шела че това е края на командата. Всичко след ";" ще бъде изпълнено сякаш се намира на следващия ред. Това прави програмата малко по-четима. Можете разбира се да сложите then на следващия ред.

Когато използваме променливи с test е добре да ги заградим с кавички. Например:

```
if [ "$name" -eq 5 ]; then
```

while ... do ... done
while оператора е условен оператор за цикъл. Най-общо казано, това което прави е "while(докато) това условие е вярно, do(изпълни) командите done ". Нека да видим следния пример:

```
#!/bin/bash
```

```
while true; do
  echo "Press CTRL-C to quit."
done
```

true в действителност е програма. Това което прави тази програма е да се изпълнява безкрайно. Използването на true се смята, че забавя вашата програма, защото шел интерпретатора първо трябва да извика програмата и след това да я изпълни. Вместо това може да използвате командата ":":

```
#!/bin/bash
while :; do
  echo "Press CTRL-C to quit."
done
```

По този начин вие постигате същия резултат, но доста по бързо. Единствения недостатък е, че програмата става по-трудно четима. Ето един по-подробен пример, който използва променливи:

```
#!/bin/bash
x=0;     # initialize x to 0
while [ "$x" -le 10 ]; do
   echo "Current value of x: $x"
   # increment the value of x:
   x=$(expr $x + 1)
   sleep 1
done
```

Както виждате използваме test (записана като квадратни скоби) за да проверим състоянието на променливата x. Опцията -le проверява дали x е по-малко(less) или равно(equal) на 10. На говорим език това се превежда по следния начин "Докато(while) x е по-малко или равно на 10, покажи текущата стойност на x, и след това добави 1 към текущата стойност на x.". sleep 1 казва на програмата да спре изпълнението си за една секунда. Както виждате това което правим тук в да проверим за равенство. Ето списък с някой опции на test:

Проверка за равенства между променливите x и y, ако променливите са числа:
x -eq y Проверява дали x е равно на y
x -ne y Проверява дали x не е равно на y
x -gt y Проверява дали x е по-голямо от y
x -lt y Проверява дали x е по-малко от y

Проверка за равенства между променливите x и y, ако променливите са текст:
x = y Проверява дали x е същитата като y
x != y Проверява дали x не е същитата като y
-n x Проверява дали x не е празен текст
-z x Проверява дали x е празен текст

От горния пример единственият ред, който може да ви се стори по-труден за рабиране е следния:

```
x=$(expr $x + 1)
```

Това което прави този ред е да увеличи стойноста на x с 1. Но какво значи $(...)? Дали е променлива? Не. На практика това е начин да кажете на шел интерпретатора, че ще изпълнявате командата expr $x + 1, и резултата от тази команда ще бъде присвоен на x. Всяка команда която бъде записана в $(...) ще бъде изпълнена:

```
#!/bin/bash
me=$(whoami)
echo "I am $me."
```

Опитайте с този пример за да разберете какво имам предвид. Горната програмка може да бъде написана по-следния начин:

```
#!/bin/bash
echo "I am $(whoami)."
```

Сами си решете кой от начините е по-лесен за вас. Има и друг начин да изпълните команда или да присвоите разултата от изпълнението на дадена команда на променлива. Този начин ще бъде обяснен по-нататък. За сега използвайте $(...).

until ... do ... done
Условния оператор until е много близък до while. Единствената разлика е, че се обръща смисъла на условието и се взима предвид новото значение. Действието на until оператора е "докато(until) това условие е вярно, изпълнявай(do) командите". Ето пример:

```
#!/bin/bash
x=0
until [ "$x" -ge 10 ]; do
   echo "Current value of x: $x"
   x=$(expr $x + 1)
   sleep 1
done
```

Този код може би ви изглежда познат. Проверете го и вижте какво прави. until ще изпълнява командите докато стойноста на променливата x е по-голяма или равна на 10. Когато стойноста на x стане 10 цикълът ще спре. Ето защо последната стойност на x която ще се изпечата е 9.

for ... in ... do ... done
for се използва кога искате да присвойте на дадена променлива набор от стойности. Например можете да напишете програма, която да изпечатва 10 точки всяка секунда:

```
#!/bin/bash
echo -n "Checking system for errors"
for dots in 1 2 3 4 5 6 7 8 9 10; do
   echo -n "."
done
echo "System clean."
```

В случай, че не знаете опцията -n на командата echo спира автоматичното добавяне на нов ред. Пробвайте командата веднъж с -n опцията и веднъж без нея за да разберете за какво става дума. Променливата dots преминава през стойностите от 1 до 10. Вижте следния пример:

```
#!/bin/bash
for x in paper pencil pen; do
   echo "The value of variable x is: $x"
   sleep 1
done
```

Когато стартирате програмата ще видите че x в началото ще има стойност paper, след което ще премине на следващата стойност, която е pencil, и след това pen. Когато свършат стойностите през който минава цикъла изпълненито му завършва.

Ето една доста полезна програма. Тя добавя .html разширение на всички файлове в текущата директория:

```
#!/bin/bash
for file in *; do
   echo "Adding .html extension to $file..."
   mv $file $file.html
   sleep 1
done
```

Ако не знаете "*" е "wild card character". Това ще рече "всичко в текущата директория", което в нашия случай представлява всички файлове в тази директория. Променливата file ще премине през всички стойности, в този случай файловете в текущата директория. След което командата mv преименува стойностите на променливата file във такива с .html разширение.

case ... in ... esac
Условния оператор case е близък до if . За предпочитане е да се използва когато имаме голям брой условия който трябва да бъдат проверени. Вземете за пример следния код:

```
#!/bin/bash
x=5     # initialize x to 5
# now check the value of x:
case $x in
  0) echo "Value of x is 0."
     ;;
  5) echo "Value of x is 5."
     ;;
  9) echo "Value of x is 9."
     ;;
  *) echo "Unrecognized value."
esac
```

Оператора case ще провери стойност на x на коя от 3-те възможности отговаря. В този случай първо ще провери дали стойноста на x е 0, след което ще провери за 5 и 9. Накая ако никое от условия не е изпълнено ще се изпечата съобщението "Unrecognized value.". Имайте предвид, че "*" означава "всичко", и в този случай това означава "която и да е стойност различна от посочените". Ако стойноста на x е различна от 0, 5, или 9, то тогава тя попада в случая "*". Когато използвате case всяко условие трябва да завършва с две ";". Може би се чудите защо да използвате case когато може да използвате if? Ето как изглежда еквивалентната програма написана с if. Вижте коя програма е по-бърза и по лесна за четене:

```
#!/bin/bash
x=5     # initialize x to 5
if [ "$x" -eq 0 ]; then
   echo "Value of x is 0."
elif [ "$x" -eq 5 ]; then
   echo "Value of x is 5."
elif [ "$x" -eq 9 ]; then
   echo "Value of x is 9."
else
   echo "Unrecognized value."

fi
```

Кавички
Кавичите играят голяма роля в шел програмирането. Има три различни вида. Те се двойни кавички: ", единична кавичка: ', и обратно наклонена кавичка: `. Различават ли се те една от друга? Да.

Обикновено използваме двойната кавичка за да обозначим с нея низ от символи и да запазим шпацията. Например, "Този низ съдържа шпации.". Низ заграден от двойни кавички се третира като един аргумент. Вземете следният скрипт за пример:

```
xconsole$ mkdir hello world
xconsole$ ls -F
hello/     world/
```

Създадохме две директории. Командата mkdir взе думите hello и world като два аргумента, създавайки по този начин две директории. А какво ще се случи сега:

```
xconsole$ mkdir "hello world"
```

```
xconsole$ ls -F
hello/     hello world/     world/
```

Създадохме една директория състояща се от две думи. Двойните кавички направиха командата да третира двете думи като един аргумент . Без тях mkdir щеше да приеме hello за първи аргумент и world за втори.

Единична кавичка се използва обикновено когато се занимаваме с променливи. Ако една променлива е заградена от двойни кавички, то нейната стойност ще бъде оценена. Но това няма да се случи ако използваме единични кавички. За да ви стане по-ясно разгледайте следният пример:

```
#!/bin/bash
x=5     # stojnosta na x e 5
# izpolzwame dwojni kawichki
echo "Using double quotes, the value of x is: $x"
# izpolzwame edinichni kawichki
echo 'Using forward quotes, the value of x is: $x'
```

Виждате ли разликата? Може да използвате двойни кавички, ако не смятате да слагата и променливи в низът който ще заграждат кавичките. Ако се чудите дали единичните кавички запазват шпациите в даден низ както това правят двойните кавички погледнете следния пример:

```
xconsole$ mkdir 'hello world'
xconsole$ ls -F
hello world/
```

Обратно наклонените кавички коренно се различават от двойните и единични кавички. Те не се използват за да запазват шпациите. Ако си спомняте по-рано използвахме следния ред:

```
x=$(expr $x + 1)
```

Както вече знаете резултата от командата expr $x + 1 се присвоява на променливата x. Същият този резултат може да бъде постигнат ако използваме обратно наклонени кавички:

```
x=`expr $x + 1`
```

Кой от начините да използвате? Този който предпочитате. Ще откриете, че обратно наклонените кавички са по-често използвани от $(...). В много случай обаче $(...) прави кода по-лесен за четене. Вземете в предвид това:

```
$!/bin/bash
echo "I am `whoami`"
```

Аритметически операции с BASH
bash ви позволява да смятате различни аритметични изрази. Както вече видяхте аритметическите операции се използват посредством командата expr. Тази команда обаче, както и командата true се смята че са доста бавни. Причината за това е че шел интерпретатора трябва всеки път да стартира true и expr командите за да ги използва. Като алтернатива на true посочихме командата ":". А като алтернатива на expr ще изпозваме следния израз $((...)). Разликата със $(...) е броя на скобите. Нека да опитаме:

```
#!/bin/bash
x=8     # stojnosta na  x e 8
y=4     # stojnosta na  y e 4
# sega shte priswojm sumata na promenliwite x i y na promenliwata z:
z=$(($x + $y))
echo "Sum na  $x + $y e $z"
```

Ако се чуствате по-комфортно с expr вместо $((...)), тогава го използвайте него.

С bash можете да събирате, изваждате, умножавате, делите числа както и да делите по модул.

Ето и техните символи:

ДЕЙСТВИЕ	ОПЕРАТОР
Събиране	+
Изваждане	-
Умножение	*
Деление	/
Деление по модул	%

Всеки от вас би трябвало да знае какво правят първите четири оператора. Ако не знаете какво означава деление по модул това е остатъка при деление на две стойности. Ето и малко bash аритметика:

```
#!/bin/bash
x=5   # initialize x to 5
y=3   # initialize y to 3

add=$(($x + $y))   # sumiraj  x sys  y i priswoj rezultata na promenliwata add
sub=$(($x - $y))   # izwadi ot  x  y i priswoj rezultata na promenliwata sub
mul=$(($x * $y))   # umnozhi  x po y i priswoj rezultata na promenliwata mul
div=$(($x / $y))   # razdeli  x na  y i priswoj rezultata na promenliwata div
mod=$(($x % $y))   # priswoj ostatyka pri delenie na  x / y na promenliwata mod

# otpechataj otgoworite:
echo "Suma: $add"
echo "Razlika: $sub"
echo "Projzwedenie: $mul"
echo "Quotient: $div"
echo "Ostatyk: $mod"
```

Отново горният код можеше да бъде написан с командата expr. Например вместо add=$(($x + $y)), щяхме да пишем add=$(expr $x + $y), или add=`expr $x + $y`.

Четене на информация от клавиатурата
Сега вече идваме към интересната част. Вие можете да направите вашите програми да си взаймодействат с потребителя и потребителя да може да си взаимодейства с програмата. Командата която ви позволява да прочетете каква стойност в въвел потребителя е read. read е вградена в bash команда която се използва съвместно с променливи, както ще видите:

```
#!/bin/bash
# gets the name of the user and prints a greeting
echo -n "Enter your name: "
read user_name
echo "Hello $user_name!"
```

Променливата тук е user_name. Разбира се може да я наречете както си искате. read ще ви изчака да въведето нещо и да натиснете клавиша ENTER. Ако не натиснете нищо, командата read ще чака докато натиснете ENTER . Ако ENTER е натиснат без да е въведено нещо то ще продължи изпълнението на програмата от следващия ред. Ето и пример:

```
#!/bin/bash
# gets the name of the user and prints a greeting
echo -n "Enter your name: "
read user_name

# the user did not enter anything:
if [ -z "$user_name" ]; then
   echo "You did not tell me your name!"
   exit
fi
```

```
echo "Hello $user_name!"
```

Ако потребителя натисне само клавиша ENTER нашата програма ще се оплаче и ще прекрати изпълнението си. В противен случай ще изпечата поздравление. Четенето на информацията която се въвежда от клавиатурата е полезно когато правите интерактивни програми, който изискват потребителя да отговори на конкретни въпроси.

Функции
Функциите правят скрипта по-лесен за поддържане. Най-общо казано фумкциите разделят програмаата на малки части. Функциите изпълняват действия, който вие сте дефинирали и може да върне стойност от изпълнението си ако желаете. Преди да продължим ще ви покажа един пример на шел програма която използва функция:

```
#!/bin/bash

# functiqta hello() samo izpechatwa syobshtenie
hello()
{
   echo "Wie ste wyw funkciq hello()"
}

echo "Izwikwame funkciqta hello()..."
# izwikwame hello() funkciqta wytre w shell skripta:
hello
echo "Weche izleznahte ot  funkciqta hello()"
```

Опитайте се да напишете тази програма и да я стартирате. Единствената цел на функцията hello() е да изпечата съобщение. Функциите естествено могат да изпълняват и по-сложни задачи. В горния пример ние извикахме функцията hello() с този ред:

```
hello
```

Когато се изпълнява този ред bash интерпретатора претърсва скрипта за ред който започва с hello(). След което открива този ред и изпълнява съдържанието на функцията.

Функциите винаги се извикват чрез тяхното име. Когато пишете функция можете да започнете функцията с function_name(), както беше направено в горния пример, или да използвате думата function т.е function function_name(). Другия начин по-който можем да започнем нашата функция е function hello():

```
function hello()
{
   echo "Wie ste wyw funkciq hello()"
}
```

Функциите винаги започват с отваряща и затваряща скоба"()", последвани от отварящи и затварящи къдрави скоби: "{...}". Тези къдрави скоби бележат началото и края на функцията. Всеки ред с код затворен в тези скоби ще бъде изпълнен и ще принадлежи единствено на функцията. Функциите трябва винаги да бъдат дефинирани преди да бъдат извикани. Нека погледнем нашата програма само, че този път ще извикаме функцията преди да е дефинирана:

```
#!/bin/bash
echo "Izwikwame funkciqta hello()..."
# call the hello() function:
hello
echo "Weche izleznahte ot  funkciqta hello()"

# function hello() just prints a message
hello()
{
```

```
   echo "Wie ste wyw funkciq hello()"
}
```

Ето какъв е резултата когато се опитаме да изпълним програмата:

```
xconsole$ ./hello.sh
Izwikwame funkciqta hello()...
./hello.sh: hello: command not found
Weche izleznahte ot  funkciqta hello()
```

Както виждате програма върна грешка. Ето защо е добре да пишете вашите функции в началото на скрипта или поне преди са ги извикате. Ето друг пример как да използваме функции:

```
#!/bin/bash
# admin.sh - administrative tool

# function new_user() creates a new user account
new_user()
{
   echo "Preparing to add a new user..."
   sleep 2
   adduser     # run the adduser program
}

echo "1. Add user"
echo "2. Exit"

echo "Enter your choice: "
read choice

case $choice in
   1) new_user     # call the new_user() function
      ;;
   *) exit
      ;;
esac
```

За да работи правилно тази програма трябва да сте влезли като root, тъй като adduser е програма която само root потребителя има право да изпълнява. Да се надяваме че този кратък пример ви е убедил в полезноста на фукциите.

Прихващане не сигнали
Може да използвате вградената команда trap за да прихващате сигнали във вашата програма. Това е добър начин да излезете нормално от програмата. Непример ако имате вървяща програма при натискането на CTRL-C ще изпратите на програмата interrupt сигнал, който ще "убие" програмата. trap ще ви позволи да прихване този сигнал и ще ви даде възможност или да продължите с изпълнението на програмата или да съобщите на потребителя, че програмата спира изпълнението си. trap има следният синтаксис:

trap dejstwie signal

dejstwie указва какво да искате да направите когато прихванете даден сигнал, а signal е сигналът който очакваме. Списък със сигналите може да откриете като пишете trap -l. Когато използвате сигнали във вашата шел програма пропуснете първите три букви на сигнала, обикновено те са SIG. Например ако сигнала за прекъсване е SIGINT, във вашата шел програма използвайте само INT. Можете да използвате и номера на сигнала. Номера на сигнала SIGINT е 2.

Пробвайте следната програма:

```
#!/bin/bash
# using the trap command

# da se zipylni funkciqta sorry() pri natiskane na CTRL-C:
trap sorry INT

# function sorry() prints a message
sorry()
{
   echo "I'm sorry Dave. I can't do that."
   sleep 3
}

# count down from 10 to 1:
for i in 10 9 8 7 6 5 4 3 2 1; do
   echo $i seconds until system failure."
   sleep 1
done
echo "System failure."
```

Сега докато програмата върви и брои числа в обратен ред натиснете CTRL-C. Това ще изпрати сигнал за прекъсване на програмата. Сигналът ще бъде прихванат от trap командата, която ще изпълни sorry() функцията. Можете да накарате trap да игнорира сигнал като поставите """ на мястото на действие. Можете да накарате trap да не прихваща сигнал като използвате "-". Например:

```
# izpylni sorry() funkciqta kogato programa poluchi signal SIGINT:
trap sorry INT

# nakaraj programata da NE prihwashta SIGINT signala :
trap - INT

# ne prawi nishto kogato se prihwane signal SIGINT:
trap " INT
```

Когато кажете на trap да не прихваща сигнала, то програма се подчинява на основното действие на сигнал, което в конкретния случай е да прекъсне програмата и да я "убие". Когато укажете trap да не прави нищо при получаване на конкретен сигнал, то програма ще продължи своето действие игнорирайки сигнала.

AND и OR

Видяхме как се използват условните оператори и колко полезни са те. Има две допълнителни неща които могат да бъдат добавени. Условните изразите с AND (или "&&") и OR (или "||"). AND условният израз изглежда по следният начин:

условие_1 && условие_2

AND изразът проверява първо най-лявото условие. Ако условието е вярно се проверява второто условие. Ако и то е вярно се изпълнява останалата част от кода на скрипта. Ако условие условие_1 не е вярно(върне резултат false), тогава условие условие_2 няма да бъде проверено. С други думи:

if(ако) условие_1 е вярно, AND(и) if(ако) условие_2 е вярно, then(тогава)...

Ето един пример с **AND** условие:

```
#!/bin/bash
x=5
y=10
if [ "$x" -eq 5 ] && [ "$y" -eq 10 ]; then
   echo "I dwete uslowiq sa wqrni."
else
   echo "Uslowiqta ne sa wqrni."
fi
```

Тук виждаме, че x и y имат стойността за която проверяваме. Променете стойноста на x от x=5 на x=12, след което пуснете отново програмата и ще се убедите, че условието не е изпълнено(връща стойност false).

OR изразът е подобен. Единствената разлика е, че проверява дали най-левият израз не е верен(т.е връща резултат false). Ако това е изпълнено се проверява следващият израз, и по-следващия:

условие_1 || условие_2

С други думи това звучи така:

if(ако) условие_1 е вярно, OR(или) ако условие_2 е вярно, тогава...

Ето защо кодът след този условен оператор ще бъде изпълнен ако поне едно от условията е вярно:

```
#!/bin/bash
x=3
y=2
if [ "$x" -eq 5 ] || [ "$y" -eq 2 ]; then
   echo "Edno ot uslowiqta e wqrno."
else
   echo "Nito edno ot uslowiqta ne e wqrno."
fi
```

В този пример ще се уверите, че едно от условията е вярно. Сменете стойността на променливата y и изпълнете отново програмата. Ще видите, че нито едно от условията не е вярно..

Ако се замислите, ще видите, че условният оператор if може да замести употребата на AND и OR изразите. Това става чрез използването на вложени if оператори. "Влагане на if оператори" означава да използваме if оператор в тялото на друг if оператор. Можете да правите влагане и на други оператори, а не само на if. Ето един пример с вложени ifоператори, който заместват използването на AND израз в кода на програмата:

```
#!/bin/bash
x=5
y=10
if [ "$x" -eq 5 ]; then
   if [ "$y" -eq 10 ]; then
      echo "I dwete uslowiq sa wqrni."
   else
      echo "Uslowiqta ne sa wqrni."
   fi
fi
```

Резултатът е същия както и ако използвахме AND израз. Проблемът е, че кодът става по трудно четим и отнема повече време за да се напише. За да се предпазите от проблеми използвайте AND и OR изрази.

Използване на аргументи
Може би сте забелязали, че повечето програми в Linux не са интерактивни. От вас се иска да въведете някакви аргументи, в противен случай получавате съобщение в което се обеснява как да използвате

програмата. Вземете за пример командата more. Ако не напишете име на файл след нея, резултатът ще бъде точно едно такова помощно съобщение. Възможно е да направите вашата шел програма да използва аргументи. За тази цел трябва да използвате специалната променлива "$#". Тази променлива съдържа общия брой на всички аргументи подадени на програмата. Например ако изпълните следната програма:

xconsole$ foo argument

$# ще има стойност 1 , защото има само един аргумент подаден на програмата. Ако имате два аргумента, тогава $# ще има стойност 2. В допълнение стойността на всеки аргумент (нулевият аргумент е винаги името на програма - foo) може да се вземе като използвате променливите $0 - за името на програмата в случая foo, $1 за стойността на първият аргумент -argument и т.н. Може да имате максимум 9 такива променливи от $0 до $9. Нека да видим това в действие:

```
#!/bin/bash
# izpechataj pyrwiq argument
# proweri dali ima pone edin argument:
if [ "$#" -ne 1 ]; then
    echo "usage: $0 "
fi

echo "Stojnosta na argumenta e $1"
```

Тази програма очаква един и само един аргумент за да тръгне. Ако я стартирате без аргументи, или подадете повече от един аргумент, програмата ще изпечата съобщение за това как да се използва. В случай че имаме само един аргумент шел програмата ще отпечата стойността на аргумента който сте подали. Припомнете си, че $0 е името на програмата. Ето защо тази специална променлива се използва в "usage" съобщението.

Пренасочване и PIPING
Обикновено, когато стартирате дадена команда, резултата от изпълнението се отпечатва на екрана. Например:

xconsole$ echo "Hello World"
Hello World

"Пренасочването" ви позволява да съхраните резултата от изпълнението накъде другаде. В повечето случаи това става към файл. Операторът ">" се използва за пренасочване на изхода. Мислете за него като за стрелка сочеща къде да отиде резултата. Ето еди пример за пренасочване на изхода към файл:

xconsole$ echo "Hello World" > foo.file
xconsole$ cat foo.file
Hello World

Тук резултатът от командата echo "Hello World" е пренасочен към файл с име foo.file. Когато прочетете съдържанието на файла ще видите там резултата. Има един проблем когато използвате оператора ">". Ако имате файл със същото име, то неговото съдържание няма да бъде запазено, а ще бъде изтрито и заместено с новото. Ами ако искате да добавите информация във файла без да изтривате старата? Тогава трябва да използвате оператора за добавяне : ">>". Използва се по същият начин с тази разлика, че не изтрива старото съдържание на файла, а го запазва и добавя новото съдържание накрая.

А сега ще ви запознаем с piping. Piping-ът ви позволява да вземете резултата от изпълнението на дадена програма и да го използвате като входни данни за друга програма. Piping става посредством оператора: "|". Забележете, че това не е малката буквата "L". Този оператор може да получите чрез натискане на клавиша SHIFT и \. Ето и един пример за piping:

xconsole$ cat /etc/passwd | grep xconsole
xconsole:x:1002:100:X_console,,,:/home/xconsole:/bin/bash

Тук четем целия файл /etc/passwd и след това резултатът е подаден за обработка на командата grep, която от своя страна, претърсва текста за низът xconsole и изпечатва целия ред съдържащ този низ на екрана. Може

да използвате и пренасочване за да запишете крайният резултат на файл:

```
xconsole$ cat /etc/passwd | grep xconsole > foo.file
xconsole$ cat foo.file
xconsole:x:1002:100:X_console,,,:/home/xconsole:/bin/bash
```

Работи. Файлът /etc/passwd е прочетен, и неговото съдържание е претърсено от командата grep за низът xconsole. След което крайният резултат е пренасочен към файл foo.file. Ще откриете, че пренасочване и piping са много полезни средства когато пишете вашите шел програми.

Временни файлове

Често ще има моменти в който ще ви се наложи да създадете временен файл. Този файл може да съдържа временна информация и просто да работи с някоя програма. В повечеето случаи с завършването на изпълнението на програмата се изтрива и временния файл. Когато създадете файл трябва да му зададете име. Проблемът е, че името на файла който създавате не трябва да съществува в директорията в която го създавате. В противен случай може да затриете важна информация. За да създадете файл с уникално име трябва да използвате "$$" символа, като представка или надставка в името на файла. Вземете за пример следния случай: искате да създадете временен файл с име hello. Има вероятност и някой друг да има файл със същото име в тази директория, което ще доведе до катастрофални резултати за вашата програма. Ако вместо това създадете фиайл с име hello.$$ или $$hello, вие ще създадете уникален файл. Опитайте:

```
xconsole$ touch hello
xconsole$ ls
hello
xconsole$ touch hello.$$
xconsole$ ls
hello     hello.689
```

Ето го и нашият временен файл.

Връщане на стойности

Повечето програми връщат стоност(и) в зависимост от начина по който завършват изпълнението си. Например, ако разгледате ръководството на командата grep, ще видите, че в него се казва че командата grep връща стойност 0 ако има съвпадение, и 1 ако не е открито съвпадение. Защо да се грижим да връщаме стойности? По много причини. Нека да кажем, че искате да проверите дали конкретно потребителско име съществува на вашата система. Единият от начините да направите това е да използвате командата grep върху файла с паролите /etc/passwd. Да предположим, че потребителското име което търсим е foobar:

```
xconsole$ grep "foobar" /etc/passwd
xconsole$
```

Няма никъв резултат от изпълнението. Това означава че grep не е намерила съвпадение. Но може да направим програмата много по-полезна ако се появява съобщение, което пояснява резултата. Това е когато искате да проверите стойноста която се връща от дадена програма. Има една специална променлива, която съдържа крайният резултат от изпълнението на програмата. Тази променлива е $?.Разгледайте следният код:

```
#!/bin/bash
# grep for user foobar and pipe all output to /dev/null:
grep "foobar" /etc/passwd > /dev/null 2>&1
# capture the return value and act accordingly:
if [ "$?" -eq 0 ]; then
   echo "Match found."
    exit
else
    echo "No match found."
fi
```

Когато стартираме програмата променливата "$?" ще прихване резултата от командата grep. Ако той е равен на 0, значи има съвпадение и подходящо съобщение ще обяви за това. В противен случай, ще изпечата, че

няма съвпадения. Това е един основен начин за получаване на резултата, който връща дадена програма. Ще откриете, че доста често ще ви се наложи да знаете стойността, която връща дадена програма за да продължите по-нататък.

Ако случайно се чудите какво значи 2>&1 сега ще ви обясня . Под Linux, тези номера обозначават файлови дескриптори. 0 е за стандартния вход (пример: клавиатура), 1 е за стандартния изход (пример: монитор) и 2 е за стандартния изход на грешките (пример: монитор). Всяка обикновена информация се изпраща на файлов дескриптор 1, и ако има грешки те се изпращат на файлов дескриптор 2. Ако не искате тези съобщения да излизат просто можете да ги пренасочите към /dev/null. Забележете, че това няма да спре изпращането на информацията на стандартния изход. Например ако нямате права да четете от директория на друг потребител вие няма да можете да видите нейното съдържание:

```
xconsole$ ls /root
ls: /root: Permission denied
xconsole$ ls /root 2> /dev/null
xconsole$
```

Както виждате съобщението за грешка не беше изпечатано. Същото важи както и за други програми така и за файлов дескриптор 1. Ако не искате резултата от изпълнението на програмата да се отпечатва на екрана, можете спокойно да го пренасочите към /dev/null. Ако не искате да виждате както резултата от изпълнението, така и съобщенията за грешка може да го направите по следният начин:

```
xconsole$ ls /root > /dev/null 2>&1
```

Това означава че резултата от програмата както и всяка грешка която предизвика тази програма ще бъдат изпратени на /dev/null, така че никога повече няма да можете да ги видите.

Какво трябва да направите ако искате вашият шел скрипт да връща стиойност при завършване на програмата? Командата exit приема само един аргумент - число което трябва да се върне при завършване на програмата. Обикновно числото 0 се използва за да кажем, че програмата е завършила успешно, т.е. не е възникнала никаква грешка по време на нейното изпълнение. Всичко по-голямо или по-малко от 0 обикновено обозначава, че е възникнала някаква грешка. Това го решавате вие като програмист. Нека проследим следната програма:

```
#!/bin/bash
if [ -f "/etc/passwd" ]; then
   echo "Password file exists."
   exit 0
else
   echo "No such file."
   exit 1
fi
```

Заключение

С това завършихме уводът в bash програмирането. Това ръководство ви дава основните знания за да можете да редактирате чужди bash скриптове или да създавате нови. За да постигнете съвършенство обаче, трябва много да практикувате. bash е идеално средство за писане на обикновени административни скриптове. Но за по-големи разработки ще се нуждаете от мощни езици като C или Perl.

Успех.

http://bg.wikibooks.org/wiki/Писане_на_скриптове_за_BASH_шел
Писане на скриптове за BASH шел

Тази статия е преведена с разрешението на автора и X_console. Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/. Както всички шелове, който може да намерите за Linux BASH (Bourne Again SHell) е не само отличен команден интерпретатор но и език за писане на криптове. Шел(Shell) скриптовете ви позволяват максимално да използвате възможностите на шел интерпретатора и да автоматизирате множество задачи. Много от програмите, който може да намерите за Linux в последно време са шел скриптове. Ако искате да разберете как те работят или как може да ги редактирате е важно да рабирате синтаксиса и семантиката на BASH шела. В допълнение познаването на bash езика ви позволява да напишете ваши собствени програми, който да изпълняват точно това което искате. Програмиране или писане на скрипове? Хората който до сега не са се занимавали с програмиране обикновено не разбират разликата между програмен и скрипт език. Програмните езици обикновено са по-мощни и по-бързи от скрипт езиците. Примери за програмни езици са C, C++, и Java. Програмите който се пишат на тези езици обикновено започват от изходен код (source code) - текст който съдържа инструкции за това как окончателната програма трябва да работи след което се компилират до изпълним файл. Тези изпълними файлове не могат лесно да бъдат адаптирани за различни операционни системи (ОС). Например ако сте написали програма на C за Linux изпълнимият файл няма да тръгне под Windows 98. За да можете да я използвате тази програма се налага да прекомпилирате изходния код под Windows 98. Скриптовете (програмите писани на скрипт езици) също започват от изходен, но не се компилират в изпълними файлове. При тях се използва интерпретатор който чете инструкциите от изходния код и ги изпълнява. За жалост, поради това че интерпретатора трябва да прочете всяка команда преди да я изпълни, интерпретираните програми вървят като цяло по-бавно спрямо компилираните. Основното предимство на скриптовете се е, че лесно могат да бъдат пренаписани за други ОС стига да има интерпретатор за тази ОС. bash е скрипт език. Той е идеален за малки програми. Други скрип езици са Perl, Lisp, и Tcl. Какво искате да знаете? Писането на собствени шел скриптове изисква от вас да знаете най-основните команди на Linux. Трябва да знаете например как да копирате, местите или създавате нови файлове. Едно от нещата което е задължително да знаете е как да работите с текстов редактор. За Linux има множество текстови редактори най-разпространените от който са vi, emacs, pico, mcedit. Внимание!!! Не се упражнявайте в писане на скриптове като root потребител! Не се знае какво може да се случи! Аз няма да бъда отговорен ако вие по невнимание повредите вашата система. Имайте това в предвид! За упражненията използвайте нормален потребител без права на root. Вашият първи BASH скрипт Първият скрипт който ще напишете е класическата "Hello World" програма. Тази програма изпечатва единствено думите "Hello World" на екрана. Отворете любимия си текстов редактор и напишете:
!/bin/bash

echo "Hello World"
Първият ред от програмата казва че ще използваме bash интерпретатора за да подкараме програмата. В този случай bash се намира в /bin директорията. Ако bash се намира в друга директория на вашата система тогава направете необходимите премени в първия ред. Изричното споменаване на това кой интерпретататор ще изпълнява скрипта е много важно, тъи като той казва на Linux какви инструкции могат да бъдат използвани в скрипта. Следващото нещо което трябва да направите е да запишете файла под името hello.sh. Остава само да направите файла изпълним. За целта пишете: xconsole$ chmod 700 ./hello.sh Прочетете упътването за използване на chmod командата ако не знаете как да променяте правата на даден файл. След като сте направили програмата изпълнима я стартирайте като напишете: xconsole$ ./hello.sh Резултатът от която ще бъде следният надпис на екрана

Hello World
Това е! Запомнете последователноста от действия - писане на кода, записване на файла с кода, и променянето на файла в изпълним с командата chmod. Команди, Команди, Команди Какво точно направи вашата първа програма? Тя изпечата думите "Hello World" на екрана. Но как програмата го направи това? С помоща на команди. Единственият ред с команди беше echo "Hello World". И коя е командата? echo. Командата echo изпечатва всичко на екрана, което е получила като свой аргумент. Аргумент е всичко което е написано след името на командата. В нашият случай това е "Hello World". Когато напишете ls /home/root, командата е ls а нейният аргимент е /home/root. Какво означава всичко това? Това означава че ако имате програма или команда, която изпечатва аргументите си на екрана то може да я използвате вместо echo. Нека да предположим че имаме такава команда която се казва foo. Тази комада ще изпечата своите аргументи на екрана. Тогава нашият скрипт може да изглежда така:
!/bin/bash

foo "Hello World" Запишете го и го направете изпълним с chmod след което го стартирайте: xconsole$ ./hello Hello World Точно същият резултат. Всичко което правихте до сега е да използвате echo командата във вашия шел скрипт. Друга команда за изпечатване е printf. Командата printf позволява повече контрол при изпечатване на информацията, особено ако сте запознати с програмния език C. Фактически същият резултат от нашият скрипт можем да постигнем просто ако напишем в командния ред: xconsole$ echo "Hello World" Hello World Както виждате може да използвате Linux команди при писането на шел скриптове. Вашият bash шел скрипт е колекция от различни програми, специално написани заедно за да изпълнят конкретна задача. Малко по-лесни програми Сега ще напишем програма с която да преместим всички файлове от дадена директория в нова директория, след което ще изтрием новата директория заедно със нейното съдържание и ще я създадем отново. Това може да бъде направено със следната последователност от команди: xconsole$ mkdir trash xconsole$ mv * trash xconsole$ rm -rf trash xconsole$ mkdir trash Вместо да пишем последователно тези команди можем да ги запишем във файл:
!/bin/bash

mkdir trash mv * trash rm -rf trash mkdir trash echo "Deleted all files!" Запишете файла под името clean.sh. След като стартирате clean.sh той ще премести всички файлове в директория trash, след което ще изтрие директорията заедно със съдържанието и ще я създаде отново. Дори ще изпечата съобщение на екрана, когато свърши със тези действия. Този пример показва и как да автоматизирате многократното писане на последователности от команди. Коментари във скрипта Коментарите ви помагат да направите вашата програма по-лесна за разбиране. Те не променят нищо в изпълнението на самата програма . Те се използват единствено за да може вие да ги четете. Всички коментари в bash започват със символа: "#", с изключение на първия ред (#!/bin/bash). Първият ред не е команда. Всички редове след първия който започват със "#" са коментари. Вижте кода на следния скрипт:
!/bin/bash
tazi programa broi chislata ot 1 do 10:

for i in 1 2 3 4 5 6 7 8 9 10; do echo $i done Дори и да не знаете bash, вие веднага може да се ориентирате какво прави скрипта, след като прочетете коментара. Добре е при писане на скриптове да използвате коментари. Ще откриете, че ако ви се наложи да промените някоя програма която сте писали преди време, коментарите ще ви бъдат от голяма полза. Променливи Променливите, най-общо казано, са "кутии" който съхраняват информация. Вие може да използвате променливи за много неща. Те ви помагат да съхранявате информацията която е въвел потребителя, аргументи или числова информация. Погледнете този код:
!/bin/bash

x=12 echo "Stoinosta na promenliwata x e $x" Всичко което се случва е да присвоим на променливата x стойност 12 и да я отпечатаме тази стойност с комадата echo. echo "Stoinosta na promenliwata x e $x" изпечатва текущата стойност на x. Когато давате стойност на дадена променлива не трябва да има шпации между нея и "=". Ето какъв е синтаксиса: име_на_променлива=стойност Стойноста на променливата можем да получим като поставим символа "$" пред нея. В конкретния случай за да изпечатаме стойноста на& x пишем echo $x. Има два типа променливи - локални променливи и променливи на обкръжението(environment). Променливите на обкръжението се задават от системата и информация за тях може да се получи като се използва env командата. Например: xconsole$ echo $SHELL Ще отпечата

/bin/bash Което е името на шела който използваме в момента. Променливите на обкръжението са дефинирани в /etc/profile и ~/.bash_profile. С echo командата можете лесно да проверите текущата стойност на дадена променлива, биля тя от обкръжението или локална. Ако все още се чудите защо са ни нужни променливи следната програма е добър пример който илюстрира тяхното използване:
!/bin/bash

echo "Stojnosta na x e 12." echo "Az imam 12 moliwa." echo "Toj mi kaza che stojnosta na x e 12." echo "Az sym na 12 godini." echo "Kak taka stojnosta na x e 12?" Да предположим че в един момент решите да промените стойноста на x от 12 на 8. Какво трябва да направите? Трябва да премените навсякъде в кода 12 с 8. Да но... има редове в който 12 не е стойноста на x. Трябва ли да променяме и тези редове? Не защото те не са свързани с x. Не е ли объркващо? По надолу е същия пример само че се използват променливи:
!/bin/bash

x=12 # stoinosta na promenliwata x e 12 e x echo "Stoinosta na x e $x." echo "Az imam 12 moliwa." echo "Toj mi kaza che stojnosta na x e $x." echo "Az sym na 12 godini." echo "Kak taka stojnosta na x e $x?" В този пример $x ще изпечата текущата стойност на x, която е 12. По този начин ако искате да промените стойноста на x от 12

на 8 е необходимо единствено да замените реда в който пише x=12 с x=8. Другите редовете няма да бъдат променени. Променливити имат и дуги полезни свойства както ще се убедите сами в последствие. Писане на скриптове за BASH шел : версия 1.2(част 2) Тази статия е преведена с разрешението на автора и X_console. Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/. Продължение на [част 1]. Условни оператори Условните оператори ви позволяват вашата програма да "взема решения" и я правят по-компактна. Което е по-важно с тях може да проверявате за грешки. Всички примери до сега започваха изпълнението си от първия ред до последния без никакви проверки. За пример:
!/bin/bash

cp /etc/foo . echo "Done." Тази малка шел програма копира файлът /etc/foo в текущата директория и изпечатва "Done" на екрана. Тази програма ще работи само при едно условие. Трябва да има файл /etc/foo. В противен случай ще се получи следния резултат: xconsole$ ./bar.sh cp: /etc/foo: No such file or directory Done. Както виждате имаме проблем. Не всеки който стартира вашата програма има файл /etc/foo на системата си. Ще бъде по-добре, ако вашата програма проверява дали файла /etc/foo съществува и ако това е така да продължи с копирането, в противен случай да спре изпълнението. С "псевдо" код това изглежда така: if /etc/code exists, then copy /etc/code to the current directory print "Done." to the screen. otherwise, print "This file does not exist." to the screen exit Може ли това да бъде направено с bash? Разбира се! В bash условните оператори са: if, while, until, for, и case. Всеки оператор започва с ключова дума и завършва с ключова дума. Например if оператора започва с ключовата дума if, и завършва с fi. Условните оператори не са програми във вашата система. Те са вградени свойства на bash. if ... else ... elif ... fi Е един от най-често използваните условни оператори. Той дава възможност на програма да вземе решения от рода на "направи това ако(if) това условие е изпълнено, или(else) прави нещо друго". За да използвате ефективно условния оператор if трябва да използвате командата test. test проверява за съществуване на файл, права, подобия или разлики. Ето програмата bar.sh:
!/bin/bash

if test -f /etc/foo then
file exists, so copy and print a message.

cp /etc/foo . echo "Done." else
file does NOT exist, so we print a message and exit.

echo "This file does not exist." exit fi Забележете че редовете след then и else са малко по-навътре. Това не е задължително, но се прави с цел програмата да бъде по-лесна за четене. Сега стартирайте програмата. Ако имате файл /etc/foo, тогава програмата ще го копира в текущата директория, в противен случай ще върне съобщение за грешка. Опцията -f проверява дали това е обикновен файл. Ето списък с опциите на командата test: -d проверява дали файлът е директория -e проверява дали файлът съществува -f проверява дали файлът е обикновен файл -g проверява дали файлът има SGID права -r проверява дали файлът може да се чете -s проверява дали файлът разнерът на файла не е 0 -u проверява дали файлът има SUID права -w проверява дали върху файлът може да се пише -x проверява дали файлът е изпълним

else се използва ако искате вашата програма да направи нещо друго, ако първото условие не е изпълнено. Има и ключова дума elif, която може да бъде използвана вместо да пишете друг if вътре в първия if. elif идва от английското "else if". Използва се когато първото условие не е изпълнено и искате да проверите за друго условие. Ако не се чувствате комфортно с if и test синтаксиса, който е : if test -f /etc/foo then тогава може да използвате следния вариант: if [-f /etc/foo]; then Квадратните скоби формират test командата. Ако имате опит в програмирането на C този синтакс може да ви се стори по-удобен. Забележете, че трябва да има разстояние след отварящата квадратна скоба и преди затварящата. Точката и запетаята: ";" казва на шела че това е края на командата. Всичко след ";" ще бъде изпълнено сякаш се намира на следващия ред. Това прави програмата малко по-четима. Можете разбира се да сложите then на следващия ред. Когато използваме променливи с test е добре да ги заградим с кавички. Например: if ["$name" -eq 5]; then while ... do ... done while оператора е условен оператор за цикъл. Най-общо казано, това което прави е "while(докато) това условие е вярно, do(изпълни) командите done ". Нека да видим следния пример:
!/bin/bash

while true; do echo "Press CTRL-C to quit." done true в действителност е програма. Това което прави тази програма е да се изпълнява безкрайно. Използването на true се смята, че забавя вашата програма, защото шел интерпретатора първо трябва да извика програмата и след това да я изпълни. Вместо това може да използвате командата ":":
!/bin/bash

while :; do echo "Press CTRL-C to quit." done По този начин вие постигате същия резултат, но доста по бързо. Единствения недостатък е, че програмата става по-трудно четима. Ето един по-подробен пример, който използва променливи:
!/bin/bash

x=0; # initialize x to 0 while ["$x" -le 10]; do echo "Current value of x: $x"
increment the value of x:

x=$(expr $x + 1) sleep 1 done Както виждате използваме test (записана като квадратни скоби) за да проверим състоянието на променливата x. Опцията -le проверява дали x е по-малко(less) или равно(equal) на 10. На говорим език това се превежда по следния начин "Докато(while) x е по-малко или равно на 10, покажи текущата стойност на x, и след това добави 1 към текущата стойност на x.". sleep 1 казва на програмата да спре изпълнението си за една секунда. Както виждате това което правим тук в да проверим за равенство. Ето списък с някой опции на test: Проверка за равенства между променливите x и y, ако променливите са числа: x -eq y Проверява дали x е равно на y x -ne y Проверява дали x не е равно на y x -gt y Проверява дали x е по-голямо от y x -lt y Проверява дали x е по-малко от y Проверка за равенства между променливите x и y, ако променливите са текст: x = y Проверява дали x е същитата като y x != y Проверява дали x не е същитата като y -n x Проверява дали x не е празен текст -z x Проверява дали x е празен текст

От горния пример единственият ред, който може да ви се стори по-труден за рабиране е следния: x=$(expr $x + 1) Това което прави този ред е да увеличи стойноста на x с 1. Но какво значи $(...)? Дали е променлива? Не. На практика това е начин да кажете на шел интерпретатора, че ще изпълнявате командата expr $x + 1, и резултата от тази команда ще бъде присвоен на x. Всяка команда която бъде записана в $(...) ще бъде изпълнена:
!/bin/bash

me=$(whoami) echo "I am $me." Опитайте с този пример за да разберете какво имам предвид. Горната програмка може да бъде написана по-следния начин:
!/bin/bash

echo "I am $(whoami)." Сами си решете кой от начините е по-лесен за вас. Има и друг начин да изпълните команда или да присвоите разултата от изпълнението на дадена команда на променлива. Този начин ще бъде обяснен по-нататък. За сега използвайте $(...). until ... do ... done Условния оператор until е много близък до while. Единствената разлика е, че се обръща смисъла на условието и се взима предвид новото значение. Действието на until оператора е "докато(until) това условие е вярно, изпълнявай(do) командите". Ето пример:
!/bin/bash

x=0 until ["$x" -ge 10]; do echo "Current value of x: $x" x=$(expr $x + 1) sleep 1 done Този код може би ви изглежда познат. Проверете го и вижте какво прави. until ще изпълнява командите докато стойноста на променливата x е по-голяма или равна на 10. Когато стойноста на x стане 10 цикълът ще спре. Ето защо последната стойност на x която ще се изпечата е 9. for ... in ... do ... done for се използва кога искате да присвойте на дадена променлива набор от стойности. Например можете да напишете програма, която да изпечатва 10 точки всяка секунда:
!/bin/bash

echo -n "Checking system for errors" for dots in 1 2 3 4 5 6 7 8 9 10; do echo -n "." done echo "System clean." В случай, че не знаете опцията -n на командата echo спира автоматичното добавяне на нов ред. Пробвайте командата веднъж с -n опцията и веднъж без нея за да разберете за какво става дума. Променливата dots преминава през стойностите от 1 до 10. Вижте следния пример:
!/bin/bash

for x in paper pencil pen; do echo "The value of variable x is: $x" sleep 1 done Когато стартирате програмата ще видите че x в началото ще има стойност paper, след което ще премине на следващата стойност, която е pencil, и след това pen. Когато свършат стойностите през който минава цикъла изпълненито му завършва. Ето една доста полезна програма. Тя добавя .html разширение на всички файлове в текущата директория:
!/bin/bash

for file in *; do echo "Adding .html extension to $file..." mv $file $file.html sleep 1 done Ако не знаете "*" е "wild card character". Това ще рече "всичко в текущата директория", което в нашия случай представлява всички

файлове в тази директория. Променливата file ще премине през всички стойности, в този случай файловете в текущата директория. След което командата mv преименува стойностите на променливата file във такива с .html разширение. case ... in ... esac Условния оператор case е близък до if . За предпочитане е да се използва когато имаме голям брой условия който трябва да бъдат проверени. Вземете за пример следния код:
!/bin/bash

x=5 # initialize x to 5
now check the value of x:

case $x in 0) echo "Value of x is 0."

5) echo "Value of x is 5."

9) echo "Value of x is 9."
) echo "Unrecognized value."

esac Оператора case ще провери стойност на x на коя от 3-те възможности отговаря. В този случай първо ще провери дали стойноста на x е 0, след което ще провери за 5 и 9. Накая ако никое от условия не е изпълнено ще се изпечата съобшението "Unrecognized value.". Имайте предвид, че "*" означава "всичко", и в този случай това означава "която и да е стойност различна от посочените". Ако стойноста на x е различна от 0, 5, или 9, то тогава тя попада в случая "*". Когато използвате case всяко условие трябва да завършва с две ";". Може би се чудите защо да използвате case когато може да използвате if? Ето как изглежда еквивалентната програма написана с if. Вижте коя програма е по-бърза и по лесна за четене:
!/bin/bash

x=5 # initialize x to 5 if ["$x" -eq 0]; then echo "Value of x is 0." elif ["$x" -eq 5]; then echo "Value of x is 5." elif ["$x" -eq 9]; then echo "Value of x is 9." else echo "Unrecognized value." fi Писане на скриптове за BASH шел : версия 1.2(част 3) Тази статия е преведена с разрешението на автора и X_console. Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/. Продължение на [част 2]. Кавички Кавичите играят голяма роля в шел програмирането. Има три различни вида: двойни кавички: ", единична кавичка: ', и обратно наклонена кавичка: `. Различават ли се те една от друга? - Да. Обикновено използваме двойната кавичка, за да обозначим с нея низ от символи и да запазим шпацията. Например, "Този низ съдържа шпации.". Низ заграден от двойни кавички се третира като един аргумент. Вземете следният скрипт за пример: xconsole$ mkdir hello world xconsole$ ls -F hello/ world/ Създадохме две директории. Командата mkdir взе думите hello и world като два аргумента, създавайки по този начин две директории. А какво ще се случи сега: xconsole$ mkdir "hello world" xconsole$ ls -F hello/ hello world/ world/ Създадохме една директория, състояща се от две думи. Двойните кавички направиха командата да третира двете думи като един аргумент . Без тях mkdir щеше да приеме hello за първи аргумент и world за втори. Единична кавичка се използва обикновено, когато се занимаваме с променливи. Ако една променлива е заградена от двойни кавички, то нейната стойност ще бъде оценена. Но това няма да се случи ако използваме единични кавички. За да ви стане по-ясно разгледайте следният пример:
!/bin/bash

x=5 # stojnosta na x e 5
izpolzwame dwojni kawichki

echo "Using double quotes, the value of x is: $x"
izpolzwame edinichni kawichki

echo 'Using forward quotes, the value of x is: $x' Виждате ли разликата? Може да използвате двойни кавички, ако не смятате да слагате и променливи в низа, който ще заграждат кавичките. Ако се чудите дали единичните кавички запазват шпациите в даден низ, както това правят двойните кавички, погледнете следния пример: xconsole$ mkdir 'hello world' xconsole$ ls -F hello world/ Обратно наклонените кавички коренно се различават от двойните и единични кавички. Те не се използват, за да запазват шпациите. Ако си спомняте по-рано използвахме следния ред: x=$(expr $x + 1) Както вече знаете резултатът от командата expr $x + 1 се присвоява на променливата x. Същият този резултат може да бъде постигнат, ако използваме обратно наклонени кавички: x=`expr $x + 1` Кой от начините да използвате? Този, който предпочитате. Ще откриете, че обратно наклонените кавички са по-често използвани от $(...). В много случай обаче $(...) прави кода по-лесен за четене. Вземете предвид това: $!/bin/bash echo "I am `whoami`" Аритметически операции с

BASH bash ви позволява да смятате различни аритметични изрази. Както вече видяхте, аритметическите операции се използват посредством командата expr. Тази команда, обаче, както и командата true се смята, че са доста бавни. Причината за това е, че шел интерпретаторът трябва всеки път да стартира true и expr командите, за да ги използва. Като алтернатива на true посочихме командата ":". А като алтернатива на expr ще използваме следния израз $((...)). Разликата със $(...) е броя на скобите. Нека да опитаме:
!/bin/bash

x=8 # stojnosta na x e 8 y=4 # stojnosta na y e 4
sega shte priswojm sumata na promenliwite x i y na promenliwata z:

z=$(($x + $y)) echo "Sum na $x + $y e $z" Ако се чуствате по-комфортно с expr вместо $((...)), тогава го използвайте него. С bash можете да събирате, изваждате, умножавате, делите числа, както и да делите по модул. Ето и техните символи: ДЕЙСТВИЕ ОПЕРАТОР Събиране + Изваждане - Умножение * Деление / Деление по модул % Всеки от Вас би трябвало да знае какво правят първите четири оператора. Ако не знаете какво означава деление по модул - това е остатъкът при деление на две стойности. Ето и малко bash аритметика:
!/bin/bash

x=5 # initialize x to 5 y=3 # initialize y to 3 add=$(($x + $y)) # sumiraj x sys y i priswoj rezultata na promenliwata add sub=$(($x - $y)) # izwadi ot x y i priswoj rezultata na promenliwata sub mul=$(($x * $y)) # umnozhi x po y i priswoj rezultata na promenliwata mul div=$(($x / $y)) # razdeli x na y i priswoj rezultata na promenliwata div mod=$(($x % $y)) # priswoj ostatyka pri delenie na x / y na promenliwata mod
otpechataj otgoworite:

echo "Suma: $add" echo "Razlika: $sub" echo "Projzwedenie: $mul" echo "Quotient: $div" echo "Ostatyk: $mod" Отново горният код можеше да бъде написан с командата expr. Например вместо add=$(($x + $y)), щяхме да пишем add=$(expr $x + $y), или add=`expr $x + $y`. Четене на информация от клавиатурата Сега вече идваме към интересната част. Вие можете да направите Вашите програми да си взаимодействат с потребителя и потребителят да може да си взаимодейства с програмата. Командата, която Ви позволява да прочетете каква стойност в въвел потребителят е read. read е вградена в bash команда, която се използва съвместно с променливи, както ще видите:
!/bin/bash
gets the name of the user and prints a greeting

echo -n "Enter your name: " read user_name echo "Hello $user_name!" Променливата тук е user_name. Разбира се, може да я наречете, както си искате. read ще Ви изчака да въведето нещо и да натиснете клавиша ENTER. Ако не натиснете нищо, командата read ще чака, докато натиснете ENTER . Ако ENTER е натиснат, без да е въведено нещо, то ще продължи изпълнението на програмата от следващия ред. Ето и пример:
!/bin/bash
gets the name of the user and prints a greeting

echo -n "Enter your name: " read user_name
the user did not enter anything:

if [-z "$user_name"]; then echo "You did not tell me your name!" exit fi echo "Hello $user_name!" Ако потребителят натисне само клавиша ENTER, нашата програма ще се оплаче и ще прекрати изпълнението си. В противен случай ще изпечата поздравление. Четенето на информацията, която се въвежда от клавиатурата е полезно, когато правите интерактивни програми, които изискват потребителят да отговори на конкретни въпроси. Функции Функциите правят скрипта по-лесен за поддържане. Най-общо казано, функциите разделят програмаата на малки части. Функциите изпълняват действия, които Вие сте дефинирали и може да върне стойност от изпълнението си ако желаете. Преди да продължим, ще Ви покажа един пример на шел програма, която използва функция:
!/bin/bash
functiqta hello() samo izpechatwa syobshtenie

hello() { echo "Wie ste wyw funkciq hello()" } echo "Izwikwame funkciqta hello()..."
izwikwame hello() funkciqta wytre w shell skripta:

hello echo "Weche izleznahte ot funkciqta hello()" Опитайте се да напишете тази програма и да я стартирате.

Единствената цел на функцията hello() е да изпечата съобщение. Функциите естествено могат да изпълняват и по-сложни задачи. В горния пример ние извикахме функцията hello() с този ред: hello Когато се изпълнява този ред bash интерпретаторът претърсва скрипта за ред, който започва с hello(). След което открива този ред и изпълнява съдържанието на функцията. Функциите винаги се извикват чрез тяхното име. Когато пишете функция, можете да започнете функцията с function_name(), както беше направено в горния пример, или да използвате думата function т.е function function_name(). Другият начин, по който можем да започнем нашата функция е function hello(): function hello() { echo "Wie ste wyw funkciq hello()" } Функциите винаги започват с отваряща и затваряща скоба"()", последвани от отварящи и затварящи къдрави скоби: "{...}". Тези къдрави скоби бележат началото и края на функцията. Всеки ред с код, затворен в тези скоби ще бъде изпълнен и ще принадлежи единствено на функцията. Функциите трябва винаги да бъдат дефинирани преди да бъдат извикани. Нека погледнем нашата програма, само че този път ще извикаме функцията преди да е дефинирана:
!/bin/bash

echo "Izwikwame funkciqta hello()..."
call the hello() function:

hello echo "Weche izleznahte ot funkciqta hello()"
function hello() just prints a message

hello() { echo "Wie ste wyw funkciq hello()" } Ето какъв е резултатът, когато се опитаме да изпълним програмата: xconsole$ ./hello.sh Izwikwame funkciqta hello()... ./hello.sh: hello: command not found Weche izleznahte ot funkciqta hello() Както виждате, програмата върна грешка. Ето защо е добре да пишете вашите функции в началото на скрипта или поне преди са ги извикате. Ето друг пример как да използваме функции:
!/bin/bash
admin.sh - administrative tool
function new_user() creates a new user account

new_user() { echo "Preparing to add a new user..." sleep 2 adduser # run the adduser program } echo "1. Add user" echo "2. Exit" echo "Enter your choice: " read choice case $choice in 1) new_user # call the new_user() function) exit

esac За да работи правилно тази програма трябва да сте влезли като root, тъй като adduser е програма, която само root потребителят има право да изпълнява. Да се надяваме, че този кратък пример Ви е убедил в полезноста на фукциите. Прихващане не сигнали Може да използвате вградената команда trap< за да прихващате сигнали във вашата програма. Това е добър начин да излезете нормално от програмата. Например, ако имате вървяща програма при натискането на CTRL-C ще изпратите на програмата interrupt сигнал, който ще "убие" програмата. trap; ще ви позволи да прихване този сигнал и ще ви даде възможност или да продължите с изпълнението на програмата, или да съобщите на потребителя, че програмата спира изпълнението си. trap има следния синтаксис: trap dejstwie signal dejstwie указва какво да искате да направите, когато прихванете даден сигнал, а signal е сигналът, който очакваме. Списък със сигналите може да откриете като пишете trap -l. Когато използвате сигнали във Вашата шел програма пропуснете първите три букви на сигнала, обикновено те са SIG. Например, ако сигналът за прекъсване е SIGINT, във Вашата шел програма използвайте само INT. Можете да използвате и номера на сигнала. Номерът на сигнала SIGINT е 2. Пробвайте следната програма:
!/bin/bash
using the trap command
da se zipylni funkciqta sorry() pri natiskane na CTRL-C:

trap sorry INT
function sorry() prints a message

sorry() { echo "I'm sorry Dave. I can't do that." sleep 3 }
count down from 10 to 1:

for i in 10 9 8 7 6 5 4 3 2 1; do echo $i seconds until system failure." sleep 1 done echo "System failure." Сега, докато програмата върви и брои числа в обратен ред, натиснете CTRL-C. Това ще изпрати сигнал за прекъсване на програмата. Сигналът ще бъде прихванат от trap командата, която ще изпълни sorry() функцията. Можете да накарате trap да игнорира сигнал като поставите "" на мястото на действие. Можете да

накарате trap да не прихваща сигнал като използвате "-". Например:
izpylni sorry() funkciqta kogato programa poluchi signal SIGINT:

trap sorry INT
nakaraj programata da NE prihwashta SIGINT signala :

trap - INT
ne prawi nishto kogato se prihwane signal SIGINT:

trap INT Когато кажете на trap да не прихваща сигнала, то програмата се подчинява на основното действие на сигнал, което в конкретния случай е да прекъсне програмата и да я "убие". Когато укажете trap да не прави нищо при получаване на конкретен сигнал, то програмата ще продължи своето действие, игнорирайки сигнала. Вижте [Част 1| Част 2 | Част 3 | Част 4] Тази статия е преведена с разрешението на автора и X_console. Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/. Продължение на [част 2]. Кавички Кавичите играят голяма роля в шел програмирането. Има три различни вида: двойни кавички: ", единична кавичка: ', и обратно наклонена кавичка: `. Различават ли се те една от друга? Да. Обикновено използваме двойната кавичка, за да обозначим с нея низ от символи и да запазим шпацията. Например, "Този низ съдържа шпации.". Низ заграден от двойни кавички се третира като един аргумент. Вземете следния скрипт за пример: xconsole$ mkdir hello world xconsole$ ls -F hello/ world/ Създадохме две директории. Командата mkdir взе думите hello и world като два аргумента, създавайки по този начин две директории. А какво ще се случи сега: xconsole$ mkdir "hello world" xconsole$ ls -F hello/ hello world/ world/ Създадохме една директория, състояща се от две думи. Двойните кавички направиха командата да третира двете думи като един аргумент . Без тях mkdir щеше да приеме hello за първи аргумент и world за втори. Единична кавичка се използва обикновено, когато се занимаваме с променливи. Ако една променлива е заградена от двойни кавички, то нейната стойност ще бъде оценена. Но това няма да се случи ако използваме единични кавички. За да Ви стане по-ясно разгледайте следния пример:
!/bin/bash

x=5 # stojnosta na x e 5
izpolzwame dwojni kawichki

echo "Using double quotes, the value of x is: $x"
izpolzwame edinichni kawichki

echo 'Using forward quotes, the value of x is: $x' Виждате ли разликата? Може да използвате двойни кавички, ако не смятате да слагате и променливи в низа, който ще заграждат кавичките. Ако се чудите дали единичните кавички запазват шпациите в даден низ, както това правят двойните кавички, погледнете следния пример: xconsole$ mkdir 'hello world' xconsole$ ls -F hello world/ Обратно наклонените кавички коренно се различават от двойните и единични кавички. Те не се използват, за да запазват шпациите. Ако си спомняте по-рано използвахме следния ред: x=$(expr $x + 1) Както вече знаете, резултатът от командата expr $x + 1 се присвоява на променливата x. Същият този резултат може да бъде постигнат ако използваме обратно наклонени кавички: x=`expr $x + 1` Кой от начините да използвате? Този, който предпочитате. Ще откриете, че обратно наклонените кавички са по-често използвани от $(...). В много случаи обаче $(...) прави кода по-лесен за четене. Вземете предвид това: $!/bin/bash echo "I am `whoami`" Аритметически операции с BASH bash ви позволява да смятате различни аритметични изрази. Както вече видяхте аритметическите операции се използват посредством командата expr. Тази команда, обаче, както и командата true се смята, че са доста бавни. Причината за това е, че шел интерпретаторът трябва всеки път да стартира true и expr командите, за да ги използва. Като алтернатива на true посочихме командата ":". А като алтернатива на expr ще изпозваме следния израз $((...)). Разликата със $(...) е броят на скобите. Нека да опитаме:
!/bin/bash

x=8 # stojnosta na x e 8 y=4 # stojnosta na y e 4
sega shte priswojm sumata na promenliwite x i y na promenliwata z:

z=$(($x + $y)) echo "Sum na $x + $y e $z" Ако се чуствате по-комфортно с expr вместо $((...)), тогава го използвайте него. С bash можете да събирате, изваждате, умножавате, делите числа, както и да делите по модул. Ето и техните символи: ДЕЙСТВИЕ ОПЕРАТОР Събиране + Изваждане - Умножение * Деление / Деление по модул % Всеки от Вас би трябвало да знае какво правят първите четири оператора. Ако не знаете какво означава деление по модул това е остатъкът при деление на две стойности. Ето и малко bash

аритметика:
!/bin/bash

x=5 # initialize x to 5 y=3 # initialize y to 3 add=$(($x + $y)) # sumiraj x sys y i priswoj rezultata na promenliwata add sub=$(($x - $y)) # izwadi ot x y i priswoj rezultata na promenliwata sub mul=$(($x * $y)) # umnozhi x po y i priswoj rezultata na promenliwata mul div=$(($x / $y)) # razdeli x na y i priswoj rezultata na promenliwata div mod=$(($x % $y)) # priswoj ostatyka pri delenie na x / y na promenliwata mod
otpechataj otgoworite:

echo "Suma: $add" echo "Razlika: $sub" echo "Projzwedenie: $mul" echo "Quotient: $div" echo "Ostatyk: $mod" Отново горният код можеше да бъде написан с командата expr. Например вместо add=$(($x + $y)), щяхме да пишем add=$(expr $x + $y), или add=`expr $x + $y`. Четене на информация от клавиатурата Сега вече идваме към интересната част. Вие можете да направите Вашите програми да си взаимодействат с потребителя и потребителят да може да си взаимодейства с програмата. Командата която Ви позволява да прочетете каква стойност в въвел потребителят е read. read е вградена в bash команда, която се използва съвместно с променливи, както ще видите:
!/bin/bash
gets the name of the user and prints a greeting

echo -n "Enter your name: " read user_name echo "Hello $user_name!" Променливата тук е user_name. Разбира се, може да я наречете, както си искате. read ще ви изчака да въведето нещо и да натиснете клавиша ENTER. Ако не натиснете нищо, командата read ще чака, докато натиснете ENTER . Ако ENTER е натиснат, без да е въведено нещо, то ще продължи изпълнението на програмата от следващия ред. Ето и пример:
!/bin/bash
gets the name of the user and prints a greeting

echo -n "Enter your name: " read user_name
the user did not enter anything:

if [-z "$user_name"]; then echo "You did not tell me your name!" exit fi echo "Hello $user_name!" Ако потребителят натисне само клавиша ENTER, нашата програма ще се оплаче и ще прекрати изпълнението си. В противен случай ще изпечата поздравление. Четенето на информацията, която се въвежда от клавиатурата е полезно, когато правите интерактивни програми, които изискват потребителят да отговори на конкретни въпроси. Функции Функциите правят скрипта по-лесен за поддържане. Най-общо казано, функциите разделят програмата на малки части. Функциите изпълняват действия, които Вие сте дефинирали и може да върне стойност от изпълнението си ако желаете. Преди да продължим, ще Ви покажа един пример на шел програма, която използва функция:
!/bin/bash
functiqta hello() samo izpechatwa syobshtenie

hello() { echo "Wie ste wyw funkciq hello()" } echo "Izwikwame funkciqta hello()..."
izwikwame hello() funkciqta wytre w shell skripta:

hello echo "Weche izleznahte ot funkciqta hello()" Опитайте се да напишете тази програма и да я стартирате. Единствената цел на функцията hello() е да изпечата съобщение. Функциите естествено могат да изпълняват и по-сложни задачи. В горния пример ние извикахме функцията hello() с този ред: hello Когато се изпълнява този ред bash интерпретатора претърсва скрипта за ред който започва с hello(). След което открива този ред и изпълнява съдържанието на функцията. Функциите винаги се извикват чрез тяхното име. Когато пишете функция можете да започнете функцията с function_name(), както беше направено в горния пример, или да използвате думата function т.е function function_name(). Другият начин, по който можем да започнем нашата функция е function hello(): function hello() { echo "Wie ste wyw funkciq hello()" } Функциите винаги започват с отваряща и затваряща скоба"()", последвани от отварящи и затварящи къдрави скоби: "{...}". Тези къдрави скоби бележат началото и края на функцията. Всеки ред с код затворен в тези скоби ще бъде изпълнен и ще принадлежи единствено на функцията. Функциите трябва винаги да бъдат дефинирани преди да бъдат извикани. Нека погледнем нашата програма, само че този път ще извикаме функцията преди да е дефинирана:
!/bin/bash

echo "Izwikwame funkciqta hello()..."

call the hello() function:

hello echo "Weche izleznahte ot funkciqta hello()"
function hello() just prints a message

hello() { echo "Wie ste wyw funkciq hello()" } Ето какъв е резултатът, когато се опитаме да изпълним програмата: xconsole$ ./hello.sh Izwikwame funkciqta hello()... ./hello.sh: hello: command not found Weche izleznahte ot funkciqta hello() Както виждате, програмата върна грешка. Ето защо е добре да пишете Вашите функции в началото на скрипта или поне преди си ги извикате. Ето друг пример как да използваме функции:
!/bin/bash
admin.sh - administrative tool
function new_user() creates a new user account

new_user() { echo "Preparing to add a new user..." sleep 2 adduser # run the adduser program } echo "1. Add user" echo "2. Exit" echo "Enter your choice: " read choice case $choice in 1) new_user # call the new_user() function
) exit

esac За да работи правилно тази програма, трябва да сте влезли като root, тъй като adduser е програма, която само root потребителят има право да изпълнява. Да се надяваме, че този кратък пример Ви е убедил в полезноста на фукциите. Прихващане на сигнали Може да използвате вградената команда trap, за да прихващате сигнали във Вашата програма. Това е добър начин да излезете нормално от програмата. Например, ако имате вървяща програма при натискането на CTRL-C ще изпратите на програмата interrupt сигнал, който ще "убие" програмата. trap ще Ви позволи да прихване този сигнал и ще Ви даде възможност или да продължите с изпълнението на програмата, или да съобщите на потребителя, че програмата спира изпълнението си. trap има следния синтаксис: trap dejstwie signal dejstwie указва какво да искате да направите, когато прихванете даден сигнал, а signal е сигналът, който очакваме. Списък със сигналите може да откриете като пишете trap -l. Когато използвате сигнали във вашата шел програма пропуснете първите три букви на сигнала, обикновено те са SIG. Например, ако сигналът за прекъсване е SIGINT, във Вашата шел програма използвайте само INT. Можете да използвате и номера на сигнала. Номерът на сигнала SIGINT е 2. Пробвайте следната програма:
!/bin/bash
using the trap command
da se zipylni funkciqta sorry() pri natiskane na CTRL-C:

trap sorry INT
function sorry() prints a message

sorry() { echo "I'm sorry Dave. I can't do that." sleep 3 }
count down from 10 to 1:

for i in 10 9 8 7 6 5 4 3 2 1; do echo $i seconds until system failure." sleep 1 done echo "System failure." Сега, докато програмата върви и брои числа в обратен ред натиснете CTRL-C. Това ще изпрати сигнал за прекъсване на програмата. Сигналът ще бъде прихванат от trap командата, която ще изпълни sorry() функцията. Можете да накарате trap да игнорира сигнал като поставите "" на мястото на действие. Можете да накарате trap да не прихваща сигнал като използвате "-". Например:
izpylni sorry() funkciqta kogato programa poluchi signal SIGINT:

trap sorry INT
nakaraj programata da NE prihwashta SIGINT signala :

trap - INT
ne prawi nishto kogato se prihwane signal SIGINT:

trap INT Когато кажете на trap да не прихваща сигнала, то програмата се подчинява на основното действие на сигнал, което в конкретния случай е да прекъсне програмата и да я "убие". Когато укажете trap да не прави нищо при получаване на конкретен сигнал, то програмата ще продължи своето действие, игнорирайки сигнала. Писане на скриптове за BASH шел : версия 1.2(част 4) Тази статия е преведена с разрешението на автора и X_console. Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/. Продължение на [част 3]. AND и OR Видяхме как се използват условните оператори и колко полезни са те. Има две

допълнителни неща, които могат да бъдат добавени. Условните изрази са AND (или "&&") и OR (или "||"). AND условният израз изглежда по следния начин: условие_1 && условие_2 AND изразът проверява първо най-лявото условие. Ако условието е вярно се проверява второто условие. Ако и то е вярно се изпълнява останалата част от кода на скрипта. Ако условие условие_1 не е вярно(върне резултат false), тогава условие условие_2 няма да бъде проверено. С други думи: if(ако) условие_1 е вярно, AND(и) if(ако) условие_2 е вярно, then(тогава)... Ето един пример с AND условие:
!/bin/bash

x=5 y=10 if ["$x" -eq 5] && ["$y" -eq 10]; then echo "I dwete uslowiq sa wqrni." else echo "Uslowiqta ne sa wqrni." fi Тук виждаме, че x и y имат стойността, за която проверяваме. Променете стойноста на x от x=5 на x=12, след което пуснете отново програмата и ще се убедите, че условието не е изпълнено(връща стойност false). OR изразът е подобен. Единствената разлика е, че проверява дали най-левият израз не е верен(т.е връща резултат false). Ако това е изпълнено се проверява следващия израз и по-следващия: условие_1 || условие_2 С други думи това звучи така: if(ако) условие_1 е вярно, OR(или) ако условие_2 е вярно, тогава... Ето защо кодът след този условен оператор ще бъде изпълнен ако поне едно от условията е вярно:
!/bin/bash

x=3 y=2 if ["$x" -eq 5] || ["$y" -eq 2]; then echo "Edno ot uslowiqta e wqrno." else echo "Nito edno ot uslowiqta ne e wqrno." fi В този пример ще се уверите, че едно от условията е вярно. Сменете стойността на променливата y и изпълнете отново програмата. Ще видите, че нито едно от условията не е вярно. Ако се замислите, ще видите, че условният оператор if може да замести употребата на AND и OR изразите. Това става чрез използването на вложени if оператори. "Влагане на if оператори" означава да използваме if оператор в тялото на друг if оператор. Можете да правите влагане и на други оператори, а не само на if. Ето един пример с вложени ifоператори, които заместват използването на AND израз в кода на програмата:
!/bin/bash

x=5 y=10 if ["$x" -eq 5]; then if ["$y" -eq 10]; then echo "I dwete uslowiq sa wqrni." else echo "Uslowiqta ne sa wqrni." fi fi Резултатът е същият, както и ако използвахме AND израз. Проблемът е, че кодът става по - трудно четим и отнема повече време, за да се напише. За да се предпазите от проблеми използвайте AND и OR изрази. Използване на аргументи Може би сте забелязали, че повечето програми в Linux не са интерактивни. От Вас се иска да въведете някакви аргументи; в противен случай получавате съобщение, в което се обяснява как да използвате програмата. Вземете за пример командата more. Ако не напишете име на файл след нея, резултатът ще бъде точно едно такова помощно съобщение. Възможно е да направите вашата шел програма да използва аргументи. За тази цел трябва да използвате специалната променлива "$#". Тази променлива съдържа общия брой на всички аргументи подадени на програмата. Например ако изпълните следната програма: xconsole$ foo argument $# ще има стойност 1 , защото има само един аргумент подаден на програмата. Ако имате два аргумента, тогава $# ще има стойност 2. В допълнение стойността на всеки аргумент (нулевият аргумент е винаги името на програма - foo) може да се вземе като използвате променливите $0 - за името на програмата в случая foo, $1 за стойноста на първият аргумент -argument и т.н. Може да имате максимум 9 такива променливи от $0 до $9. Нека да видим това в действие:
!/bin/bash
izpechataj pyrwiq argument
proweri dali ima pone edin argument:

if ["$#" -ne 1]; then echo "usage: $0 " fi echo "Stojnosta na argumenta e $1" Тази програма очаква един и само един аргумент, за да тръгне. Ако я стартирате без аргументи, или подадете повече от един аргумент, програмата ще изпечата съобщение за това как да се използва. В случай, че имаме само един аргумент шел програмата ще отпечата стойноста на аргумента, който сте подали. Припомнете си, че $0 е името на програмата. Ето защо тази специална променлива се използва в "usage" съобщението. Пренасочване и PIPING Обикновено, когато стартирате дадена команда, резултатът от изпълнението се отпечатва на екрана. Например: xconsole$ echo "Hello World" Hello World "Пренасочването" Ви позволява да съхраните резултата от изпълнението някъде другаде. В повечето случаи това става към файл. Операторът ">" се използва за пренасочване на изхода. Мислете за него като за стрелка, сочеща къде да отиде резултата. Ето един пример за пренасочване на изхода към файл: xconsole$ echo "Hello World" > foo.file xconsole$ cat foo.file Hello World Тук резултатът от командата echo "Hello World" е пренасочен към файл с име foo.file. Когато прочетете съдържанието на файла ще видите там резултата. Има един проблем, когато използвате оператора ">". Ако имате файл със същото име, то неговото съдържание няма да бъде запазено, а ще бъде изтрито и заместено с новото. Ами ако искате да добавите информация във файла, без да изтривате старата? Тогава трябва да използвате операторът за добавяне : ">>". Използва се по същия начин с тази разлика, че не изтрива старото

съдържание на файла, а го запазва и добавя новото съдържание накрая. А сега ще ви запознаем с piping. Piping-ът Ви позволява да вземете резултата от изпълнението на дадена програма и да го използвате като входни данни за друга програма. Piping става посредством оператора: "|". Забележете, че това не е малката буквата "L". Този оператор може да получите чрез натискане на клавиша SHIFT и \. Ето и един пример за piping: xconsole$ cat /etc/passwd | grep xconsole xconsole:x:1002:100:X_console,,,:/home/xconsole:/bin/bash Тук четем целия файл /etc/passwd и след това резултатът е подаден за обработка на командата grep, която от своя страна претърсва текста за низа xconsole и изпечатва целия ред, съдържащ този низ на екрана. Може да използвате и пренасочване, за да запишете крайния резултат на файл: xconsole$ cat /etc/passwd | grep xconsole > foo.file xconsole$ cat foo.file xconsole:x:1002:100:X_console,,,:/home/xconsole:/bin/bash Работи. Файлът /etc/passwd е прочетен и неговото съдържание е претърсено от командата grep за низа xconsole. След което крайният резултат е пренасочен към файл foo.file. Ще откриете, че пренасочване и piping са много полезни средства, когато пишете Вашите шел програми. Временни файлове Често ще има моменти, в които ще Ви се наложи да създадете временен файл. Този файл може да съдържа временна информация и просто да работи с някоя програма. В повечето случаи със завършването на изпълнението на програмата се изтрива и временният файл. Когато създадете файл трябва да му зададете име. Проблемът е, че името на файла, който създавате не трябва да съществува в директорията, в която го създавате. В противен случай може да затриете важна информация. За да създадете файл с уникално име трябва да използвате "$$" символа като представка или надставка в името на файла. Вземете за пример следния случай: искате да създадете временен файл с име hello. Има вероятност и някой друг да има файл със същото име в тази директория, което ще доведе до катастрофални резултати за Вашата програма. Ако вместо това създадете фиайл с име hello.$$ или $$hello, Вие ще създадете уникален файл. Опитайте: xconsole$ touch hello xconsole$ ls hello xconsole$ touch hello.$$ xconsole$ ls hello hello.689 Ето го и нашият временен файл. Връщане на стойности Повечето програми връщат стойност(и) в зависимост от начина, по който завършват изпълнението си. Например, ако разгледате ръководството на командата grep ще видите, че в него се казва, че командата grep връща стойност 0 ако има съвпадение, и 1 ако не е открито съвпадение. Защо да се грижим да връщаме стойности? По много причини. Нека да кажем, че искате да проверите дали конкретно потребителско име съществува на Вашата система. Единият от начините да направите това е да използвате командата grep върху файла с паролите /etc/passwd. Да предположим, че потребителското име, което търсим е foobar: xconsole$ grep "foobar" /etc/passwd xconsole$ Няма никакъв резултат от изпълнението. Това означава че grep не е намерила съвпадение. Но може да направим програмата много по-полезна ако се появява съобщение, което пояснява резултата. Това е, когато искате да проверите стойноста, която се връща от дадена програма. Има една специална променлива, която съдържа крайния резултат от изпълнението на програмата. Тази променлива е $?.Разгледайте следния код:

```
!/bin/bash
grep for user foobar and pipe all output to /dev/null:

grep "foobar" /etc/passwd > /dev/null 2>&1
capture the return value and act accordingly:

if [ "$?" -eq 0 ]; then echo "Match found." exit else echo "No match found." fi
```

Когато стартираме програмата, променливата "$?" ще прихване резултата от командата grep. Ако той е равен на 0, значи има съвпадение и подходящо съобщение ще обяви за това. В противен случай ще изпечата, че няма съвпадения. Това е един основен начин за получаване на резултата, който връща дадена програма. Ще откриете, че доста често ще Ви се наложи да знаете стойността, която връща дадена програма, за да продължите по-нататък. Ако случайно се чудите какво значи 2>&1, сега ще ви обясня . Под Linux, тези номера обозначават файлови дескриптори. 0 е за стандартния вход (пример: клавиатура), 1 е за стандартния изход (пример: монитор) и 2 е за стандартния изход на грешките (пример: монитор). Всяка обикновена информация се изпраща на файлов дескриптор 1, и ако има грешки те се изпращат на файлов дескриптор 2. Ако не искате тези съобщения да излизат просто можете да ги пренасочите към /dev/null. Забележете, че това няма да спре изпращането на информацията на стандартния изход. Например, ако нямате права да четете от директория на друг потребител, Вие няма да можете да видите нейното съдържание: xconsole$ ls /root ls: /root: Permission denied xconsole$ ls /root 2> /dev/null xconsole$ Както виждате, съобщението за грешка не беше изпечатано. Същото важи както за други програми, така и за файлов дескриптор 1. Ако не искате резултатът от изпълнението на програмата да се отпечатва на екрана, можете спокойно да го пренасочите към /dev/null. Ако не искате да виждате както резултатът от изпълнението, така и съобщенията за грешка, може да го направите по следния начин: xconsole$ ls /root > /dev/null 2>&1 Това означава че резултатът от програмата, както и всяка грешка, която предизвика тази програма ще бъдат изпратени на /dev/null, така че никога повече няма да можете да ги видите. Какво трябва да направите ако искате Вашият шел скрипт да връща стойност при завършване на програмата? Командата exit приема само един аргумент - число, което трябва да се върне при завършване на

програмата. Обикновно числото 0 се използва, за да кажем, че програмата е завършила успешно, т.е. не е възникнала никаква грешка по време на нейното изпълнение. Всичко по-голямо или по-малко от 0 обикновено обозначава, че е възникнала някаква грешка. Това го решавате Вие като програмист. Нека проследим следната програма:
!/bin/bash

if [-f "/etc/passwd"]; then echo "Password file exists." exit 0 else echo "No such file." exit 1 fi Заключение С това завършихме увода в bash програмирането. Това ръководство Ви дава основните знания, за да можете да редактирате чужди bash скриптове или да създавате нови. За да постигнете съвършенство обаче, трябва много да практикувате. bash е идеално средство за писане на обикновени административни скриптове. Но за по-големи разработки ще се нуждаете от мощни езици като C или Perl. Успех!

http://bg.wikipedia.org/wiki/Bash
Bash или Bourne Again Shell (игра на думи: „възродена обвивка“ или „отново обвивка на Борн“) е команден интерпретатор (обвивка), използван предимно в подобните на UNIX операционни системи.
Съдържание

Въведение

В системите на основата на UNIX, командният интерпретатор изпълнява функцията на преводач между потребителя и ядрото на операционната система. Дълго време това е бил основният и най-добър начин за работа с UNIX. За разлика от създадените много по-късно системи с графичен интерфейс като Windows и MacOS, работата с командните интерпретатори се осъществява посредством текстови команди. Те могат да бъдат външни (в отделни изпълними файлове) или вградени в интерпретатора (builtins).
Предшественици

Един от първите командни интерпретатори за UNIX е Bourne shell (sh). Той носи името си от своя създател Стивън Борн и е бил включен в UNIX Версия 7 през 1979 година. Друг широко разпространен интерпретатор е C shell (csh), написан от Бил Джой в калифорнийския университет Бъркли като част от операционната система BSD. Със синтаксис подобен на езика C, C shell е изключително удобен и лесен за научаване от UNIX програмисти. За съжаление той е несъвместим с Bourne shell. Първият съществен опит за подобряване на възможностите на Bourne shell е интерпретаторът ksh (Korn Shell). Той съчетава нови възможности и съвместимост с Bourne shell.
Роден отново

Създаден като част от проекта ГНУ, Bash (Bourne-Again Shell) е един от най-популярните командни интерпретатори в UNIX. Базира се на небезизвестния sh (Bourne Shell) като също така взаимства функционалност от ksh (Korn Shell) и csh (C Shell). Официално е започнат през 10 януари 1988 г. от Брайън Фокс към който по-късно се присъединява и Чет Рами. През 1995 г. Чет започва работа над Bash 2.0 официално обявен на 23 декември 1996 г. Bash посредством своя синтаксис и вградени команди предоставя възможност за програмиране. Той може да изпълнява команди както от командния ред, така и от външен файл. Синтаксисът му е един и същ независимо от къде чете командите. Мощта на програмирането на Bash не се състои толкова в ключовите му думи и конструкции, колкото в набора от десетките програми като sed, awk, grep, cut и др. които са незаменима част от UNIX.

Навигация в Bash

Съществуват някои основни принципи в навигацията на шела, които след заучаването важат и за множество текстови редактори и други *nix програми. За командите man и less, както и vi важат следните клавиши.
J -- предвижване на долу
K -- предвижване на горе
L -- предвижване на дясно
H -- предвижване на ляво
Space -- една страница на долу
q --- от (quit) излизане от режима
Ctrl + Z -- скриване , преустановяване на текущата работа (job) на "фона"
fg -- връщане към скритата работа от "фона"
clear ; anotherCommand --- две или повече команди могат да бъдат изпълнени на един ред ако съществува ; помежду им
Ctrl + C --- спиране изпълнението на текуща команда

Примери за синтаксиса на Bash скриптове

Преди да започване редакцията на файл, които ще представлява изпълним скрипт съдържащ поредица от команди на Операционната система трябва да се окаже изпълнимостта му. Т.е. ако името на файла е myFirstScript.sh , следната команда изпълнена в директорията на скрипта ще го направи изпълним:
chmod u+x myFirstScript.sh

Това може да бъде лесно проверено с командата ls -al

В случай, че се зарежда изпълним файл които съдържа Bash код, в началото му трябва да се укаже пътя към интерпретатора по следния начин:
#!/път/към/bash

За разлика от езици като C, C++, Java и др. при Bash кода не се компилира, а се интерпретира така както е подаден. Типичен пример за една Bash програма е следният:
```
#!/usr/local/bin/bash
echo "Здравей, Свят!"
```

Кодът може да се коментира със # например:
```
echo "hello" # Отпечатай hello
# прекъсни програмата
exit
```

В Bash Може да създавате променливи и функции. Типа на променливите не е задължителен, но при нужда може да се укаже посредством вградената команда declare. Променлива се декларира по следния начин:
ime_promenliva="stoinost na promenlivata"

И се достига със знакът за стринг $ в началото и. Например:
echo $ime_promenliva

Има няколко начина за създаване на фунции. Единият е чрез ключовата дума function:
```
function ime_funciq {
  # kod na funciq
}
```

Или със скоби по следния начин:
```
ime_funciq() {
  # kod na funciq
}
```

Функциите се извикват само с името им, без скоби или допълнителни символи в началото, например:
ime_funciq "параметър 1" "параметър 2" три четири 5

Параметрите подадени на функцията се разполагат от $1 до $n където n е число и е последен параметър. Те могат да се изместват с вградената команда shift, например:
```
echo $1
shift 2
# $3 става $1
```
Логика и изражения в синтаксиса Bash
Цифрови сравнения [редактиране]

За простота е използвано често използваното съкращение int , което произлиза от англ. integer --- цяло число:
int1 -eq int2 Връща True ако int1 е равно на int2
int1 -ge int2 Връща True ако int1 е по-голямо или равно на int2.
int1 -gt int2 Връща True ако int1 е по-голямо от int2.
int1 -le int2 Връща True ако int1 е по-малко или равно от int2
int1 -lt int2 Връща True ако int1 е по-малко от int2
int1 -ne int2 Връща True ако int1 не е равно на int2
Сравнения на низове

На английски string, което се съкращава str означава низ.
str1 = str2 Връща True ако str1 е идентична на str2.
str1 != str2 Връща True ако str1 не е идентична на str2.
str Връща True ако str не е null (т.е. празен)
-n str Връща True if дължината на str е по-голяма от null
(null означава липса на стойност или нищо)
-z str Връща True ако дължината на str е равна на 0 . (0 е различна от null)
Примери за команди изпълнени под Bash [редактиране]
ls -a -X -1 # листване на всички файлове вертикално с всичките им изпълними атрибути
ls -al | less # същото със стопиране на екрана
history | grep cd # пример за пренасочване на изходен поток от данни от командата history , която показва последните .... команди изпълнени в шела в програмата grep , която филтрира и показва редовете съдържащи cd низа
!345 # изпълнение на команда номер 345 от историята на командите
command > command_output.txt # създаване на файла command_output.txt от изходният поток (резултата на командатата command)
find / -name "filename" # намери файл с името "filename" от започвайки търсенето рекурсивно от / директорията
find . -name linux | grep bash | less # намери всички файлове които съдържат низа "linux" в името си , филтрирай ги така , че покажи само редовете съдържащи низа "bash" и филтрирай последно през less командата за по-лесно виждане
alias ls='ls -1 -X --color=tty' # създаване на алиаз
Ctrl + A # отиване в началото на реда
Ctrl + E # отиване в края на реда
Ctrl + U # изтриване на текста наляво от курсора
Ctrl + K # изтриване на текста надясно от курсора
mkdir /mnt/usbflash # създаване на /mnt/usbflash директорията
mount /dev/sdb1 -t vfat /mnt/usbflash # свързване на преносим носител за данни с vfat тип файлова система
umount /mnt/usbflash # отвързване на файловата система
cat history.txt | mail -s "test file sending" -c someEmail(a)abv.bg some.otherEmail.com # изпращане на файлът history.txt , чрез програмата mail с заглавие "test file sending" до посочените адреси
head -n 20 tooLongFile # показване на първите 20 реда от файл
vim /etc/bashrc # промяна на промпта
PS1="\u@\h \t \w\n\\$ " # различен вид промпт от този под подразбиране
$ for file in `ls -R` ; do cp $file $file.bak; done # копирай всеки един файл под текущата директория с именафайл.bak
gunzip *file.zip
tar -xvf file.tar # разпакетиране на ципиран файл

Променяне на bash промпта

Преди да започнем да променяме bash промпта е добре да запазим старата му стойност, за всеки случай. Стойността на промпта се пази от променливата на обкръжението PS1. За да запазим тази стойност е добре да я присвоим на друга променлива. Това става по следният начин:

```
[slaff@atlantis slaff]$ SAVE=$PS1
```

Сега старата стойност на PS1 променливата се пази от променливата SAVE. PS1 променливата определя как да изглежда нашият промпт. Нека да направим първата промяна и да присвоим на PS1 променливата стойност "command>":

```
[slaff@atlantis slaff]$ PS1="command>"
```

В резултат на тази опреция ще имаме следния промпт:

```
command>
```

Сега нека се опитаме да възстановим стария промпт

```
command> PS1=$SAVE
[slaff@atlantis slaff]$
```

Bash позволява използването и на специални символи, който да стоят в нашият промпт. Ето и някой от тези специални символи:

\a предизвиква пиукане на спикера
\d показва дата във формат "Ден от седмицата" "Месец" Ден от месеца" (примерно "Tue May 26")
\h името на хоста до първата точка (пр. хост subdomain.domain.org -> името на хоста до първата точка = subdomain)
\H цялото име на хоста
\n нов ред
\r нов ред
\s името на шелът, който използваме
\t времето за 24 часа в следния формат HH:MM:SS (HH-час, MM-минути, SS-секунди)
\T време за 12 часа в следния формат HH:MM:SS
\u потребителското име (username)
\v версия на bash шела(примерно 2.00)
\w пълният път до текущата директория
\W само името на текущата директория
\! коя подред е тази команда
\nnn осмично число
\\ обратно наклонена черта (т.е. \)
\[начало на последователност от "контролни символи"
\] край на последователност от "контролни символи"

Промптът, който използвахме до момента е съставен от следните специални символи:

\u (потребителско име) т.е slaff
\h името на хоста до първата точка т.е atlantis
\W името на текущата директория т.е slaff

```
[slaff@atlantis slaff]$ PS1="\u@\h \W> "
slaff@atlantis slaff> ls
bin mail
slaff@atlantis slaff>
```

Този промпт е най-често използван в повечето Linux дистибуции.
Нека сега да променим промпта така, че да показва и часът:

```
slaff@atlantis slaff> PS1="[\t][\u@\h:\w]\$ "
[21:52:01][slaff@atlantis:~]$ ls
bin mail
```

Ако не искаме всеки път да променяме промта, а той да се променя още с влизането ни, трябва да променим файлът .bash_profile, който се намира във нашата главна директория. Ако такъв файл не съществува трябва да го създадем. Ако имате root права и искате да промените на всички потребители промпта редактирайте файлът /etc/profile или /etc/bashrc . Имайте предвид, че този файл може да се намира на друго място за различните Linux дистрибуции. Самото редактране се състои в добавянето на този ред
PS1="[\t][\u@\h:\w]\$ "

Добавяне на псевдоними (alias)

Alias представлява псевдоним(алтернативно име) на дадена команда или последователности от команди и се използва за улеснение. Ако например искате всеки път вместо да пишете командата "cd /usr/local/share/" да напишете само GO трябва да направите следния псевдоним

```
alias GO="cd /usr/local/share/"
```

За да не го пишете всеки път този псевдним е най-добре да го добавите във файлът .bash_profile

Променливата PROMPT_COMMAND

Прменливата PROMPT_COMMAND се показва или изпълнява преди да се покаже самият промпт.
Разгледайте следният пример:

```
[21:55:01][slaff@atlantis:~] PS1="[\u@\h:\w]\$ "
[slaff@atlantis:~] PROMPT_COMMAND="echo 2155"
2155
[slaff@atlantis:~] d
bin mail
2156
[slaff@atlantis:~]
```

Това което се случва е да се показва числото 2155 винаги преди показването на самият ред. Проблемът тук е, че 2155 не се показава на същия ред на който се показва и промпта. За целта ще използваме опцията -n на командата echo.

```
[slaff@atlantis:~] PROMPT_COMMAND="echo -n 2155"
2156[slaff@atlantis:~]$
2156[slaff@atlantis:~]$ d
bin mail
```

Вече нещата изглеждат по-добре.

Сега ще ви покажа как можете да накарате промта да показва размера който заема текущата директория. Запишете следния Bash скрипт в /usr/local/bin директорията под името lsbytesum. Директория може и да е друга, стига да е посочена в променливата PATH.

```
#!/bin/bash
# lsbytesum - sum the number of bytes in a directory listing
TotalBytes=0
for Bytes in $(ls -l | grep "^-" | cut -c30-41)
do
let TotalBytes=$TotalBytes+$Bytes
done
TotalMeg=$(echo -e "scale=3 \n$TotalBytes/1048576 \nquit" | bc)
echo -n "$TotalMeg"
```

Направете този файл изпълним.

Променяме промпта:

```
[2158][slaff@atlantis:~]$ PS1="[\u@\h:\w (\$(lsbytesum) Mb)]\$ "
[slaff@atlantis:~ (0 Mb)]$ cd /bin
[slaff@atlantis:/bin (4.498 Mb)]$
```

Това е. Вече имаме промпт, който показава колко е големината на текущата директория.
Нищо не пречи да се създадат и други промптове. Опитайте сами да създадете нов.

Цветове за Bash

Може би сте забелязали цветните съобщения при стартирането на RedHat.
Писането на цветни съобщения е елементарно. Нека да си направим наше цветно съобщение. Опитайте следния пример:

```
echo -e "\033[41;33;1m Welcome to Linux\033[m"
```

Тази команда ще изпечата "Welcome to Linux" с жълти букви на червен фон. Забележете, че текста е обграден от "\033[bg;fg;1m \033[m". Числото 41 указва, че фонът на буквите трябва да е червен, а 33 указва, че цвета на буквите трябва да е жълт.

Кодовете на цветовете са следните:

За цвят на фона: 30=черно, 31=червено, 32=зелено, 33=жълто, 34=синьо, 35=пурпурно, 36=тюркоазено, 37=бяло
За цвят на буквите: 0=прозрачно, 40=черно, 41=червено, 42=зелено, 43=жълто, 44=синьо 45=пурпурно, 46=тюркоазено, 47=бяло

Ако искаме да променим нашия надпис и да го направим с бели букви на син фон трябва да напишем следното

```
echo -e "\033[44;37;1m Welcome to Linux\033[m"
```

Ето и една примерна програма за Slackware дистрибуцията на Linux

```
#/bin/bash
version="`cat /etc/slackware-version`"
echo -e "\033[41;33;1m Welcome to Slackware $version\033[m"
```

Освен цветни съобщения може да имате и цветен промпт. Оцветяването на промпта става по същия начин както и нормален текст с тази разлика, че трябва да заградим текста с \[\033[и m\]. За да получим светло син промпт трябва да направим следното

```
PS1="\[\033[1;34m\][\$(date +%H%M)][\u@\h:\w]$\[\033[0m\] "
```

Ето и кодовете за цвят на буквите:

Черно 0;30 Тъмно сиво 1;30
Синьо 0;34 Светло синьо 1;34
Зелено 0;32 Светло зелено 1;32
Цикламено 0;36 Светло цикламено 1;36
Червено 0;31 Светло червено 1;31
Пурпурно0;35 Светло пурпурно 1;35
Кафяво 0;33 Жълто 1;33
Светло сиво 0;37 Бяло 1;37

Ако искате вашият промпт да бъде със светло синьо на червен фон трябва да зиползвате следната комбинация:

```
\[\033[44;1;31m\]
```

или

```
\[\033[44m\]\[\033[1;31m\]
```

Допълнителни кодове:
4: Подчертан, 5: Мигащ, 8: Неведидим

Ето и един примерен скрипт

```
function elite
{

local GRAY="\[\033[1;30m\]"
local LIGHT_GRAY="\[\033[0;37m\]"
local CYAN="\[\033[0;36m\]"
local LIGHT_CYAN="\[\033[1;36m\]"

case $TERM in
xterm*)
local TITLEBAR='\[\033]0;\u@\h:\w\007\]'
;;
*)
local TITLEBAR=""
;;
esac

local GRAD1=$(tty|cut -d/ -f3)
PS1="$TITLEBAR\
$GRAY-$CYAN-$LIGHT_CYAN(\
$CYAN\u$GRAY@$CYAN\h\
$LIGHT_CYAN)$CYAN-$LIGHT_CYAN(\
$CYAN\#$GRAY/$CYAN$GRAD1\
$LIGHT_CYAN)$CYAN-$LIGHT_CYAN(\
$CYAN\$(date +%H%M)$GRAY/$CYAN\$(date +%d-%b-%y)\
$LIGHT_CYAN)$CYAN-$GRAY-\
$LIGHT_GRAY\n\
$GRAY-$CYAN-$LIGHT_CYAN(\
$CYAN\$$GRAY:$CYAN\w\
$LIGHT_CYAN)$CYAN-$GRAY-$LIGHT_GRAY "
PS2="$LIGHT_CYAN-$CYAN-$GRAY-$LIGHT_GRAY "
}
```

Не е много лесно да се помнят кодовете на отделните цветове затова може да си помогнете със следния скрипт:

```
#!/bin/bash
#
# This file echoes a bunch of colour codes to the terminal to demonstrate
# what's available. Each line is one colour on black and gray
# backgrounds, with the code in the middle. Verified to work on white,
# black, and green BGs (2 Dec 98).
#
echo " On Light Gray: On Black:"
echo -e "\033[47m\033[1;37m White \033[0m\
1;37m \
\033[40m\033[1;37m White \033[0m"
echo -e "\033[47m\033[37m Light Gray \033[0m\
37m \
\033[40m\033[37m Light Gray \033[0m"
echo -e "\033[47m\033[1;30m Gray \033[0m\
1;30m \
\033[40m\033[1;30m Gray \033[0m"
echo -e "\033[47m\033[30m Black \033[0m\
30m \
\033[40m\033[30m Black \033[0m"
echo -e "\033[47m\033[31m Red \033[0m\
31m \
\033[40m\033[31m Red \033[0m"
echo -e "\033[47m\033[1;31m Light Red \033[0m\
1;31m \
\033[40m\033[1;31m Light Red \033[0m"
echo -e "\033[47m\033[32m Green \033[0m\
32m \
\033[40m\033[32m Green \033[0m"
echo -e "\033[47m\033[1;32m Light Green \033[0m\
1;32m \
\033[40m\033[1;32m Light Green \033[0m"
echo -e "\033[47m\033[33m Brown \033[0m\
33m \
\033[40m\033[33m Brown \033[0m"
echo -e "\033[47m\033[1;33m Yellow \033[0m\
1;33m \
\033[40m\033[1;33m Yellow \033[0m"
echo -e "\033[47m\033[34m Blue \033[0m\
34m \
\033[40m\033[34m Blue \033[0m"
echo -e "\033[47m\033[1;34m Light Blue \033[0m\
1;34m \
\033[40m\033[1;34m Light Blue \033[0m"
echo -e "\033[47m\033[35m Purple \033[0m\
35m \
\033[40m\033[35m Purple \033[0m"
echo -e "\033[47m\033[1;35m Pink \033[0m\
1;35m \
\033[40m\033[1;35m Pink \033[0m"
echo -e "\033[47m\033[36m Cyan \033[0m\
36m \
\033[40m\033[36m Cyan \033[0m"
```

```
echo -e "\033[47m\033[1;36m Light Cyan \033[0m\
1;36m \
\033[40m\033[1;36m Light Cyan \033[0m"
```

Примери

По надолу е посочен доста интересен промпт който си сменя цветовете в зависимост от натоварването на системата.

```
#!/bin/bash
# "hostloadcolour" - 17 October 98, by Giles
#
# The idea here is to change the colour of the host name in the промпт,
# depending on a threshold load value.

# THRESHOLD_LOAD is the value of the one minute load (multiplied
# by one hundred) at which you want
# the промпт to change from COLOUR_LOW to COLOUR_HIGH
THRESHOLD_LOAD=200
COLOUR_LOW='1;34'
# light blue
COLOUR_HIGH='1;31'
# light red

function промпт_command {
ONE=$(uptime | sed -e "s/.*load average: \(.*\...\), \(.*\...\), \(.*\...\)/\1/" -e "s/ //g")
# Apparently, "scale" in bc doesn't apply to multiplication, but does
# apply to division.
ONEHUNDRED=$(echo -e "scale=0 \n $ONE/0.01 \nquit \n" | bc)
if [ $ONEHUNDRED -gt $THRESHOLD_LOAD ]
then
HOST_COLOUR=$COLOUR_HIGH
# Light Red
else
HOST_COLOUR=$COLOUR_LOW
# Light Blue
fi
}

function hostloadcolour {

PROMPT_COMMAND=промпт_command
PS1="[$(date +%H%M)][\u@\[\033[\$(echo -n \$HOST_COLOUR)m\]\h\[\033[0m\]:\w]$ "
}
```

Писане на скриптове за BASH шел : версия 1.2(част 1)
от X_console(14-03-2000)

Тази статия е преведена с разрешението на автора и X_console.
Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/.

Както всички шелове, който може да намерите за Linux BASH (Bourne Again SHell) е не само отличен команден интерпретатор но и език за писане на криптове. Шел(Shell) скриптовете ви позволяват максимално да използвате възможностите на шел интерпретатора и да автоматизирате множество задачи. Много от програмите, който може да намерите за Linux в последно време са шел скриптове. Ако искате да разберете как те работят или как може да ги редактирате е важно да рабирате синтаксиса и семантиката на BASH шела. В допълнение познаването на bash езика ви позволява да напишете ваши собствени програми, който да изпълняват точно това което искате.

Програмиране или писане на скрипове?
Хората който до сега не са се занимавали с програмиране обикновено не разбират разликата между програмен и скрипт език. Програмните езици обикновено са по-мощни и по-бързи от скрипт езиците. Примери за програмни езици са C, C++, и Java. Програмите който се пишат на тези езици обикновено започват от изходен код (source code) - текст който съдържа инструкции за това как окончателната програма трябва да работи след което се компилират до изпълним файл. Тези изпълними файлове не могат лесно да бъдат адаптирани за различни операционни системи (ОС). Например ако сте написали програма на C за Linux изпълнимият файл няма да тръгне под Windows 98. За да можете да я използвате тази програма се налага да прекомпилирате изходния код под Windows 98. Скриптовете (програмите писани на скрипт езици) също започват от изходен, но не се компилират в изпълними файлове. При тях се изполва интерпретатор който чете инструкциите от изходния код и ги изпълнява. За жалост, поради това че интерпретатора трябва да прочете всяка команда преди да я изпълни, интерпретираните програми вървят като цяло по-бавно спрямо компилираните. Основното предимство на скриптовете се е, че лесно могат да бъдат пренаписани за други ОС стига да има интерпретатор за тази ОС. bash е скрипт език. Той е идеален за малки програми. Други скрип езици са Perl, Lisp, и Tcl.

Какво искате да знаете?

Писането на собствени шел скриптове изисква от вас да знаете най-основните команди на Linux. Трябва да знаете например как да копирате, местите или създавате нови файлове. Едно от нещата което е задължително да знаете е как да работите с текстов редактор. За Linux има множество текстови редактори най-разпространените от който са vi, emacs, pico, mcedit.

Внимание!!!
Не се упражнявайте в писане на скриптове като root потребител! Не се знае какво може да се случи! Аз няма да бъда отговорен ако вие по невнимание повредите вяшата система. Имайте това в предвид! За упражненията изполвайте нормален потребител без права на root.

Вашият първи BASH скрипт
Първият скрипт който ще напишете е класическата "Hello World" програма. Тази програма изпечатва единствено думите "Hello World" на екрана. Отворете любимия си текстов редактор и напишете:

```
#!/bin/bash
echo "Hello World"
```

Първият ред от програмата казва че ще използваме bash интерпретатора за да подкараме програмата. В този случай bash се намира в /bin директорията. Ако bash се намира в друга директория на вашата система тогава направете необходимите премени в първия ред. Изричното споменаване на това кой интерпретататор ще изпълнява скрипта е много важно, тъи като той казва на Linux какви инструкции могат да бъдат използвани в скрипта. Следващото нещо което трябва да направите е да запишете файла под името hello.sh. Остава само да направите файла изпълним. За целта пишете:

```
xconsole$ chmod 700 ./hello.sh
```

Прочетете упътването за използване на chmod командата ако не знаете как да променяте правата на даден

файл. След като сте направили програмата изпълнима я стартирайте като напишете:

xconsole$ ./hello.sh

Резултатът от която ще бъде следният надпис на екрана

Hello World

Това е! Запомнете последователноста от действия - писане на кода, записване на файла с кода, и променянето на файла в изпълним с командата chmod.

Команди, Команди, Команди
Какво точно направи вашата първа програма? Тя изпечата думите "Hello World" на екрана. Но как програмата го направи това? С помоща на команди. Единственият ред с команди беше echo "Hello World". И коя е командата? echo. Командата echo изпечатва всичко на екрана, което е получила като свой аргумент.

Аргумент е всичко което е написано след името на командата. В нашият случай това е "Hello World". Когато напишете ls /home/root, командата е ls а нейният аргимент е /home/root. Какво означава всичко това? Това означава че ако имате програма или команда, която изпечатва аргументите си на екрана то може да я използвате вместо echo. Нека да предположим че имаме такава команда която се казва foo. Тази комада ще изпечата своите аргументи на екрана. Тогава нашият скрипт може да изглежда така:

```
#!/bin/bash
foo "Hello World"
```

Запишете го и го направете изпълним с chmod след което го стартирайте:

```
xconsole$ ./hello
Hello World
```

Точно същият резултат. Всичко което правихте до сега е да използвате echo командата във вашия шел скрипт. Друга команда за изпечатване е printf. Командата printf позволява повече контрол при изпечатване на информацията, особено ако сте запознати с програмния език C. Фактически същият резултат от нашият скрипт можем да постигнем просто ако напишем в командния ред:

```
xconsole$ echo "Hello World"
Hello World
```

Както виждате може да използвате Linux команди при писането на шел скриптове. Вашият bash шел скрипт е колекция от различни програми, специално написани заедно за да изпълнят конкретна задача.

Малко по-лесни програми
Сега ще напишем програма с която да преместим всички файлове от дадена директория в нова директория, след което ще изтрием новата директория заедно със нейното съдържание и ще я създадем отново. Това може да бъде направено със следната последователност от команди:

```
xconsole$ mkdir trash
xconsole$ mv * trash
xconsole$ rm -rf trash
xconsole$ mkdir trash
```

Вместо да пишем последователно тези команди можем да ги запишем във файл:

```
#!/bin/bash
mkdir trash
mv * trash
rm -rf trash
mkdir trash
```

```
echo "Deleted all files!"
```

Запишете файла под името clean.sh. След като стартирате clean.sh той ще премести всички файлове в директория trash, след което ще изтрие директорията заедно със съдържанието и ще я създаде отново. Дори ще изпечата съобщение на екрана, когато свърши със тези действия. Този пример показва и как да автоматизирате многократното писане на последователности от команди.

Коментари във скрипта
Коментарите ви помагат да направите вашата програма по-лесна за разбиране. Те не променят нищо в изпълнението на самата програма . Те се използват единствено за да може вие да ги четете. Всички коментари в bash започват със символа: "#", с изключение на първия ред (#!/bin/bash). Първият ред не е команда. Всички редове след първия който започват със "#" са коментари. Вижте кода на следния скрипт:

```
#!/bin/bash
# tazi programa broi chislata ot 1 do 10:
for i in 1 2 3 4 5 6 7 8 9 10; do
    echo $i
done
```

Дори и да не знаете bash, вие веднага може да се ориентирате какво прави скрипта, след като прочетете коментара. Добре е при писане на скриптове да използвате коментари. Ще откриете, че ако ви се наложи да промените някоя програма която сте писали преди време, коментарите ще ви бъдат от голяма полза.

Променливи
Променливите, най-общо казано, са "кутии" който съхраняват информация. Вие може да използвате променливи за много неща. Те ви помагат да съхранявате информацията която е въвел потребителя, аргументи или числова информация. Погледнете този код:

```
#!/bin/bash
x=12
echo "Stoinosta na promenliwata x e $x"
```

Всичко което се случва е да присвоим на променливата x стойност 12 и да я отпечатаме тази стойност с комадата echo. echo "Stoinosta na promenliwata x e $x" изпечатва текущата стойност на x. Когато давате стойност на дадена променлива не трябва да има шпации между нея и "=". Ето какъв е синтаксиса:

```
име_на_променлива=стойност
```

Стойноста на променливата можем да получим като поставим символа "$" пред нея. В конкретния случай за да изпечатаме стойноста на& x пишем echo $x.

Има два типа променливи - локални променливи и променливи на обкръжението(environment). Променливите на обкръжението се задават от системата и информация за тях може да се получи като се използва env командата. Например:

```
xconsole$ echo $SHELL
```

Ще отпечата

```
/bin/bash
```

Което е името на шела който използваме в момента. Променливите на обкръжението са дефинирани в /etc/profile и ~/.bash_profile. С echo командата можете лесно да проверите текущата стойност на дадена променлива, биля тя от обкръжението или локална. Ако все още се чудите защо са ни нужни променливи следната програма е добър пример който илюстрира тяхното използване:

```
#!/bin/bash
echo "Stojnosta na x e 12."
echo "Az imam 12 moliwa."
```

```
echo "Toj mi kaza che stojnosta na x e 12."
echo "Az sym na 12 godini."
echo "Kak taka stojnosta na x e 12?"
```

Да предположим че в един момент решите да промените стойноста на x от 12 на 8. Какво трябва да направите? Трябва да премените навсякъде в кода 12 с 8. Да но... има редове в който 12 не е стойноста на x. Трябва ли да променяме и тези редове? Не защото те не са свързани с x. Не е ли объркващо? По надолу е същия пример само че се използват променливи:

```
#!/bin/bash
x=12     # stoinosta na promenliwata x e 12 e x
echo "Stoinosta na  x e $x."
echo "Az imam 12 moliwa."
echo "Toj mi kaza che stojnosta na x e $x."
echo "Az sym na 12 godini."
echo "Kak taka stojnosta na x e $x?"
```

В този пример $x ще изпечата текущата стойност на x, която е 12. По този начин ако искате да промените стойноста на x от 12 на 8 е необходимо единствено да замените реда в който пише x=12 с x=8. Другите редовете няма да бъдат променени. Променливити имат и дуги полезни свойства както ще се убедите сами в последствие.

[край на част 1]
Писане на скриптове за BASH шел : версия 1.2(част 2)

от X_console(29-03-2000)

рейтинг (1) [добре] [зле]

Вариант за отпечатване

Тази статия е преведена с разрешението на автора и X_console.
Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/.

Продължение на [част 1].

Условни оператори
Условните оператори ви позволяват вашата програма да "взема решения" и я правят по-компактна. Което е по-важно с тях може да проверявате за грешки. Всички примери до сега започваха изпълнението си от първия ред до последния без никакви проверки. За пример:

```
#!/bin/bash
cp /etc/foo .
echo "Done."
```

Тази малка шел програма копира файлът /etc/foo в текущата директория и изпечатва "Done" на екрана. Тази програма ще работи само при едно условие. Трябва да има файл /etc/foo. В противен случай ще се получи следния резултат:

```
xconsole$ ./bar.sh
cp: /etc/foo: No such file or directory
Done.
```

Както виждате имаме проблем. Не всеки който стартира вашата програма има файл /etc/foo на системата си. Ще бъде по-добре, ако вашата програма проверява дали файла /etc/foo съществува и ако това е така да продължи с копирането, в противен случай да спре изпълнението. С "псевдо" код това изглежда така:

```
if /etc/code exists, then
   copy /etc/code to the current directory
   print "Done." to the screen.
otherwise,
   print "This file does not exist." to the screen
   exit
```

Може ли това да бъде направено с bash? Разбира се! В bash условните оператори са: if, while, until, for, и case. Всеки оператор започва с ключова дума и завършва с ключова дума. Например if оператора започва с ключовата дума if, и завършва с fi. Условните оператори не са програми във вашата система. Те са вградени свойства на bash.

if ... else ... elif ... fi
Е един от най-често използваните условни оператори. Той дава възможност на програма да вземе решения от рода на "направи това ако(if) това условие е изпълнено, или(else) прави нещо друго". За да използвате ефективно условния оператор if трябва да използвате командата test. test проверява за съществуване на файл, права, подобия или разлики. Ето програмата bar.sh:

```
#!/bin/bash
if test -f /etc/foo
then
   # file exists, so copy and print a message.
   cp /etc/foo .
   echo "Done."
else
   # file does NOT exist, so we print a message and exit.
   echo "This file does not exist."
   exit
fi
```

Забележете че редовете след then и else са малко по-навътре. Това не е задължително, но се прави с цел програмата да бъде по-лесна за четене. Сега стартирайте програмата. Ако имате файл /etc/foo, тогава програмата ще го копира в текущата директория, в противен случай ще върне съобщение за грешка. Опцията -f проверява дали това е обикновен файл. Ето списък с опциите на командата test:

-d проверява дали файлът е директория
-e проверява дали файлът съществува
-f проверява дали файлът е обикновен файл
-g проверява дали файлът има SGID права
-r проверява дали файлът може да се чете
-s проверява дали файлът разнерът на файла не е 0
-u проверява дали файлът има SUID права
-w проверява дали върху файлът може да се пише
-x проверява дали файлът е изпълним

else се използва ако искате вашата програма да направи нещо друго, ако първото условие не е изпълнено. Има и ключова дума elif, която може да бъде използвана вместо да пишете друг if вътре в първия if. elif идва от английското "else if". Използва се когато първото условие не е изпълнено и искате да проверите за друго условие.

Ако не се чувствате комфортно с if и test синтаксиса, който е :

```
if test -f /etc/foo
then
```

тогава може да използвате следния вариант:

```
if [ -f /etc/foo ]; then
```

Квадратните скоби формират test командата. Ако имате опит в програмирането на C този синтакс може да ви се стори по-удобен. Забележете, че трябва да има разстояние след отварящата квадратна скоба и преди затварящата. Точката и запетаята: ";" казва на шела че това е края на командата. Всичко след ";" ще бъде изпълнено сякаш се намира на следващия ред. Това прави програмата малко по-четима. Можете разбира се да сложите then на следващия ред.

Когато използваме променливи с test е добре да ги заградим с кавички. Например:

```
if [ "$name" -eq 5 ]; then
```

while ... do ... done
while оператора е условен оператор за цикъл. Най-общо казано, това което прави е "while(докато) това условие е вярно, do(изпълни) командите done ". Нека да видим следния пример:

```
#!/bin/bash
while true; do
  echo "Press CTRL-C to quit."
done
```

true в действителност е програма. Това което прави тази програма е да се изпълнява безкрайно. Използването на true се смята, че забавя вашата програма, защото шел интерпретатора първо трябва да извика програмата и след това да я изпълни. Вместо това може да използвате командата ":":

```
#!/bin/bash
while :; do
  echo "Press CTRL-C to quit."
done
```

По този начин вие постигате същия резултат, но доста по бързо. Единствения недостатък е, че програмата става по-трудно четима. Ето един по-подробен пример, който използва променливи:

```
#!/bin/bash
x=0;     # initialize x to 0
while [ "$x" -le 10 ]; do
   echo "Current value of x: $x"
   # increment the value of x:
   x=$(expr $x + 1)
   sleep 1
done
```

Както виждате използваме test (записана като квадратни скоби) за да проверим състоянието на променливата x. Опцията -le проверява дали x е по-малко(less) или равно(equal) на 10. На говорим език това се превежда по следния начин "Докато(while) x е по-малко или равно на 10, покажи текущата стойност на x, и след това добави 1 към текущата стойност на x.". sleep 1 казва на програмата да спре изпълнението си за една секунда. Както виждате това което правим тук в да проверим за равенство. Ето списък с някой опции на test:

Проверка за равенства между променливите x и y, ако променливите са числа:
x -eq y Проверява дали x е равно на y
x -ne y Проверява дали x не е равно на y
x -gt y Проверява дали x е по-голямо от y
x -lt y Проверява дали x е по-малко от y

Проверка за равенства между променливите x и y, ако променливите са текст:
x = y Проверява дали x е същитата като y
x != y Проверява дали x не е същитата като y
-n x Проверява дали x не е празен текст
-z x Проверява дали x е празен текст

От горния пример единственият ред, който може да ви се стори по-труден за рабиране е следния:

x=$(expr $x + 1)

Това което прави този ред е да увеличи стойноста на x с 1. Но какво значи $(...)? Дали е променлива? Не. На практика това е начин да кажете на шел интерпретатора, че ще изпълнявате командата expr $x + 1, и резултата от тази команда ще бъде присвоен на x. Всяка команда която бъде записана в $(...) ще бъде изпълнена:

```
#!/bin/bash
me=$(whoami)
echo "I am $me."
```

Опитайте с този пример за да разберете какво имам предвид. Горната програмка може да бъде написана по-следния начин:

```
#!/bin/bash
echo "I am $(whoami)."
```

Сами си решете кой от начините е по-лесен за вас. Има и друг начин да изпълните команда или да присвоите разултата от изпълнението на дадена команда на променлива. Този начин ще бъде обяснен по-нататък. За сега използвайте $(...).

until ... do ... done
Условния оператор until е много близък до while. Единствената разлика е, че се обръща смисъла на условието и се взима предвид новото значение. Действието на until оператора е "докато(until) това условие е вярно, изпълнявай(do) командите". Ето пример:

```
#!/bin/bash
x=0
until [ "$x" -ge 10 ]; do
   echo "Current value of x: $x"
   x=$(expr $x + 1)
   sleep 1
done
```

Този код може би ви изглежда познат. Проверете го и вижте какво прави. until ще изпълнява командите докато стойноста на променливата x е по-голяма или равна на 10. Когато стойноста на x стане 10 цикълът ще спре. Ето защо последната стойност на x която ще се изпечата е 9.

for ... in ... do ... done
for се използва кога искате да присвойте на дадена променлива набор от стойности. Например можете да напишете програма, която да изпечатва 10 точки всяка секунда:

```
#!/bin/bash
echo -n "Checking system for errors"
for dots in 1 2 3 4 5 6 7 8 9 10; do
   echo -n "."
done
echo "System clean."
```

В случай, че не знаете опцията -n на командата echo спира автоматичното добавяне на нов ред. Пробвайте командата веднъж с -n опцията и веднъж без нея за да разберете за какво става дума. Променливата dots преминава през стойностите от 1 до 10. Вижте следния пример:

```
#!/bin/bash
for x in paper pencil pen; do
   echo "The value of variable x is: $x"
   sleep 1
```

```
done
```

Когато стартирате програмата ще видите че x в началото ще има стойност paper, след което ще премине на следващата стойност, която е pencil, и след това pen. Когато свършат стойностите през който минава цикъла изпълненито му завършва.

Ето една доста полезна програма. Тя добавя .html разширение на всички файлове в текущата директория:

```
#!/bin/bash
for file in *; do
    echo "Adding .html extension to $file..."
    mv $file $file.html
    sleep 1
done
```

Ако не знаете "*" е "wild card character". Това ще рече "всичко в текущата директория", което в нашия случай представлява всички файлове в тази директория. Променливата file ще премине през всички стойности, в този случай файловете в текущата директория. След което командата mv преименува стойностите на променливата file във такива с .html разширение.

case ... in ... esac
Условния оператор case е близък до if . За предпочитане е да се използва когато имаме голям брой условия който трябва да бъдат проверени. Вземете за пример следния код:

```
#!/bin/bash
x=5     # initialize x to 5
# now check the value of x:
case $x in
  0) echo "Value of x is 0."
     ;;
  5) echo "Value of x is 5."
     ;;
  9) echo "Value of x is 9."
     ;;
  *) echo "Unrecognized value."
esac
```

Оператора case ще провери стойност на x на коя от 3-те възможности отговаря. В този случай първо ще провери дали стойноста на x е 0, след което ще провери за 5 и 9. Накая ако никое от условия не е изпълнено ще се изпечата съобшението "Unrecognized value.". Имайте предвид, че "*" означава "всичко", и в този случай това означава "която и да е стойност различна от посочените". Ако стойноста на x е различна от 0, 5, или 9, то тогава тя попада в случая "*". Когато използвате case всяко условие трябва да завършва с две ";". Може би се чудите защо да използвате case когато може да използвате if? Ето как изглежда еквивалентната програма написана с if. Вижте коя програма е по-бърза и по лесна за четене:

```
#!/bin/bash
x=5     # initialize x to 5
if [ "$x" -eq 0 ]; then
    echo "Value of x is 0."
elif [ "$x" -eq 5 ]; then
    echo "Value of x is 5."
elif [ "$x" -eq 9 ]; then
    echo "Value of x is 9."
else
    echo "Unrecognized value."
fi
```

Писане на скриптове за BASH шел : версия 1.2(част 3)
от X_console(4-05-2000)
Тази статия е преведена с разрешението на автора и X_console.
Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/.

Продължение на [част 2].

Кавички
Кавичите играят голяма роля в шел програмирането. Има три различни вида: двойни кавички: ", единична кавичка: ', и обратно наклонена кавичка: `. Различават ли се те една от друга? - Да.

Обикновено използваме двойната кавичка, за да обозначим с нея низ от символи и да запазим шпацията. Например, "Този низ съдържа шпации.". Низ заграден от двойни кавички се третира като един аргумент. Вземете следният скрипт за пример:

```
xconsole$ mkdir hello world
xconsole$ ls -F
hello/ world/
```

Създадохме две директории. Командата mkdir взе думите hello и world като два аргумента, създавайки по този начин две директории. А какво ще се случи сега:

```
xconsole$ mkdir "hello world"
xconsole$ ls -F
hello/ hello world/ world/
```

Създадохме една директория, състояща се от две думи. Двойните кавички направиха командата да третира двете думи като един аргумент . Без тях mkdir щеше да приеме hello за първи аргумент и world за втори.

Единична кавичка се използва обикновено, когато се занимаваме с променливи. Ако една променлива е заградена от двойни кавички, то нейната стойност ще бъде оценена. Но това няма да се случи ако използваме единични кавички. За да ви стане по-ясно разгледайте следният пример:

```
#!/bin/bash
x=5 # stojnosta na x e 5
# izpolzwame dwojni kawichki
echo "Using double quotes, the value of x is: $x"
# izpolzwame edinichni kawichki
echo 'Using forward quotes, the value of x is: $x'
```

Виждате ли разликата? Може да използвате двойни кавички, ако не смятате да слагате и променливи в низа, който ще заграждат кавичките. Ако се чудите дали единичните кавички запазват шпациите в даден низ, както това правят двойните кавички, погледнете следния пример:

```
xconsole$ mkdir 'hello world'
xconsole$ ls -F
hello world/
```

Обратно наклонените кавички коренно се различават от двойните и единични кавички. Те не се използват, за да запазват шпациите. Ако си спомняте по-рано използвахме следния ред:

```
x=$(expr $x + 1)
```

Както вече знаете резултатът от командата expr $x + 1 се присвоява на променливата x. Същият този резултат може да бъде постигнат, ако използваме обратно наклонени кавички:

```
x=`expr $x + 1`
```

Кой от начините да използвате? Този, който предпочитате. Ще откриете, че обратно наклонените кавички са

по-често използвани от $(...). В много случай обаче $(...) прави кода по-лесен за четене. Вземете предвид това:

```
$!/bin/bash
echo "I am `whoami`"
```

Аритметически операции с BASH
bash ви позволява да смятате различни аритметични изрази. Както вече видяхте, аритметическите операции се използват посредством командата expr. Тази команда, обаче, както и командата true се смята, че са доста бавни. Причината за това е, че шел интерпретаторът трябва всеки път да стартира true и expr командите, за да ги използва. Като алтернатива на true посочихме командата ":". А като алтернатива на expr ще изпозваме следния израз $((...)). Разликата със $(...) е броя на скобите. Нека да опитаме:

```
#!/bin/bash
x=8 # stojnosta na x e 8
y=4 # stojnosta na y e 4
# sega shte priswojm sumata na promenliwite x i y na promenliwata z:
z=$(($x + $y))
echo "Sum na $x + $y e $z"
```

Ако се чуствате по-комфортно с expr вместо $((...)), тогава го използвайте него.

С bash можете да събирате, изваждате, умножавате, делите числа, както и да делите по модул. Ето и техните символи:

ДЕЙСТВИЕ	ОПЕРАТОР
Събиране	+
Изваждане	-
Умножение	*
Деление	/
Деление по модул	%

Всеки от Вас би трябвало да знае какво правят първите четири оператора. Ако не знаете какво означава деление по модул - това е остатъкът при деление на две стойности. Ето и малко bash аритметика:

```
#!/bin/bash
x=5 # initialize x to 5
y=3 # initialize y to 3

add=$(($x + $y)) # sumiraj x sys y i priswoj rezultata na promenliwata add
sub=$(($x - $y)) # izwadi ot x y i priswoj rezultata na promenliwata sub
mul=$(($x * $y)) # umnozhi x po y i priswoj rezultata na promenliwata mul
div=$(($x / $y)) # razdeli x na y i priswoj rezultata na promenliwata div
mod=$(($x % $y)) # priswoj ostatyka pri delenie na x / y na promenliwata mod

# otpechataj otgoworite:
echo "Suma: $add"
echo "Razlika: $sub"
echo "Projzwedenie: $mul"
echo "Quotient: $div"
echo "Ostatyk: $mod"
```

Отново горният код можеше да бъде написан с командата expr. Например вместо add=$(($x + $y)), щяхме да пишем add=$(expr $x + $y), или add=`expr $x + $y`.

Четене на информация от клавиатурата
Сега вече идваме към интересната част. Вие можете да направите Вашите програми да си взаимодействат с потребителя и потребителят да може да си взаимодейства с програмата. Командата, която Ви позволява да прочетете каква стойност в въвел потребителят е read. read е вградена в bash команда, която се използва

съвместно с променливи, както ще видите:

```
#!/bin/bash
# gets the name of the user and prints a greeting
echo -n "Enter your name: "
read user_name
echo "Hello $user_name!"
```

Променливата тук е user_name. Разбира се, може да я наречете, както си искате. read ще Ви изчака да въведето нещо и да натиснете клавиша ENTER. Ако не натиснете нищо, командата read ще чака, докато натиснете ENTER . Ако ENTER е натиснат, без да е въведено нещо, то ще продължи изпълнението на програмата от следващия ред. Ето и пример:

```
#!/bin/bash
# gets the name of the user and prints a greeting
echo -n "Enter your name: "
read user_name

# the user did not enter anything:
if [ -z "$user_name" ]; then
echo "You did not tell me your name!"
exit
fi

echo "Hello $user_name!"
```

Ако потребителят натисне само клавиша ENTER, нашата програма ще се оплаче и ще прекрати изпълнението си. В противен случай ще изпечата поздравление. Четенето на информацията, която се въвежда от клавиатурата е полезно, когато правите интерактивни програми, които изискват потребителят да отговори на конкретни въпроси.

Функции

Функциите правят скрипта по-лесен за поддържане. Най-общо казано, функциите разделят програмаата на малки части. Функциите изпълняват действия, които Вие сте дефинирали и може да върне стойност от изпълнението си ако желаете. Преди да продължим, ще Ви покажа един пример на шел програма, която използва функция:

```
#!/bin/bash

# functiqta hello() samo izpechatwa syobshtenie
hello()
{
echo "Wie ste wyw funkciq hello()"
}

echo "Izwikwame funkciqta hello()..."
# izwikwame hello() funkciqta wytre w shell skripta:
hello
echo "Weche izleznahte ot funkciqta hello()"
```

Опитайте се да напишете тази програма и да я стартирате. Единствената цел на функцията hello() е да изпечата съобщение. Функциите естествено могат да изпълняват и по-сложни задачи. В горния пример ние извикахме функцията hello() с този ред:

```
hello
```

Когато се изпълнява този ред bash интерпретаторът претърсва скрипта за ред, който започва с hello(). След което открива този ред и изпълнява съдържанието на функцията.

Функциите винаги се извикват чрез тяхното име. Когато пишете функция, можете да започнете функцията с function_name(), както беше направено в горния пример, или да използвате думата function т.е function function_name(). Другият начин, по който можем да започнем нашата функция е function hello():

```
function hello()
{
echo "Wie ste wyw funkciq hello()"
}
```

Функциите винаги започват с отваряща и затваряща скоба"()", последвани от отварящи и затварящи къдрави скоби: "{...}". Тези къдрави скоби бележат началото и края на функцията. Всеки ред с код, затворен в тези скоби ще бъде изпълнен и ще принадлежи единствено на функцията. Функциите трябва винаги да бъдат дефинирани преди да бъдат извикани. Нека погледнем нашата програма, само че този път ще извикаме функцията преди да е дефинирана:

```
#!/bin/bash
echo "Izwikwame funkciqta hello()..."
# call the hello() function:
hello
echo "Weche izleznahte ot funkciqta hello()"

# function hello() just prints a message
hello()
{
echo "Wie ste wyw funkciq hello()"
}
```

Ето какъв е резултатът, когато се опитаме да изпълним програмата:

```
xconsole$ ./hello.sh
Izwikwame funkciqta hello()...
./hello.sh: hello: command not found
Weche izleznahte ot funkciqta hello()
```

Както виждате, програмата върна грешка. Ето защо е добре да пишете вашите функции в началото на скрипта или поне преди са ги извикате. Ето друг пример как да използваме функции:

```
#!/bin/bash
# admin.sh - administrative tool

# function new_user() creates a new user account
new_user()
{
echo "Preparing to add a new user..."
sleep 2
adduser # run the adduser program
}

echo "1. Add user"
echo "2. Exit"

echo "Enter your choice: "
read choice

case $choice in
1) new_user # call the new_user() function
;;
*) exit
;;
```

esac

За да работи правилно тази програма трябва да сте влезли като root, тъй като adduser е програма, която само root потребителят има право да изпълнява. Да се надяваме, че този кратък пример Ви е убедил в полезноста на фукциите.

Прихващане не сигнали
Може да използвате вградената команда trap< за да прихващате сигнали във вашата програма. Това е добър начин да излезете нормално от програмата. Например, ако имате вървяща програма при натискането на CTRL-C ще изпратите на програмата interrupt сигнал, който ще "убие" програмата. trap; ще ви позволи да прихване този сигнал и ще ви даде възможност или да продължите с изпълнението на програмата, или да съобщите на потребителя, че програмата спира изпълнението си. trap има следния синтаксис:

trap dejstwie signal

dejstwie указва какво да искате да направите, когато прихванете даден сигнал, а signal е сигналът, който очакваме. Списък със сигналите може да откриете като пишете trap -l. Когато използвате сигнали във Вашата шел програма пропуснете първите три букви на сигнала, обикновено те са SIG. Например, ако сигналът за прекъсване е SIGINT, във Вашата шел програма използвайте само INT. Можете да използвате и номера на сигнала. Номерът на сигнала SIGINT е 2. Пробвайте следната програма:

```
#!/bin/bash
# using the trap command

# da se zipylni funkciqta sorry() pri natiskane na CTRL-C:
trap sorry INT

# function sorry() prints a message
sorry()
{
echo "I'm sorry Dave. I can't do that."
sleep 3
}

# count down from 10 to 1:
for i in 10 9 8 7 6 5 4 3 2 1; do
echo $i seconds until system failure."
sleep 1
done
echo "System failure."
```

Сега, докато програмата върви и брои числа в обратен ред, натиснете CTRL-C. Това ще изпрати сигнал за прекъсване на програмата. Сигналът ще бъде прихванат от trap командата, която ще изпълни sorry() функцията. Можете да накарате trap да игнорира сигнал като поставите """ на мястото на действие. Можете да накарате trap да не прихваща сигнал като използвате "-". Например:

```
# izpylni sorry() funkciqta kogato programa poluchi signal SIGINT:
trap sorry INT

# nakaraj programata da NE prihwashta SIGINT signala :
trap - INT

# ne prawi nishto kogato se prihwane signal SIGINT:
trap " INT
```

Когато кажете на trap да не прихваща сигнала, то програмата се подчинява на основното действие на сигнал, което в конкретния случай е да прекъсне програмата и да я "убие". Когато укажете trap да не прави нищо при получаване на конкретен сигнал, то програмата ще продължи своето действие, игнорирайки сигнала.

Тази статия е преведена с разрешението на автора и X_console.
Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/.

Продължение на [част 2].

Кавички
Кавичите играят голяма роля в шел програмирането. Има три различни вида: двойни кавички: ", единична кавичка: ', и обратно наклонена кавичка: `. Различават ли се те една от друга? Да.

Обикновено използваме двойната кавичка, за да обозначим с нея низ от символи и да запазим шпацията. Например, "Този низ съдържа шпации.". Низ заграден от двойни кавички се третира като един аргумент. Вземете следния скрипт за пример:

```
xconsole$ mkdir hello world
xconsole$ ls -F
hello/ world/
```

Създадохме две директории. Командата mkdir взе думите hello и world като два аргумента, създавайки по този начин две директории. А какво ще се случи сега:

```
xconsole$ mkdir "hello world"
xconsole$ ls -F
hello/ hello world/ world/
```

Създадохме една директория, състояща се от две думи. Двойните кавички направиха командата да третира двете думи като един аргумент . Без тях mkdir щеше да приеме hello за първи аргумент и world за втори.

Единична кавичка се използва обикновено, когато се занимаваме с променливи. Ако една променлива е заградена от двойни кавички, то нейната стойност ще бъде оценена. Но това няма да се случи ако използваме единични кавички. За да Ви стане по-ясно разгледайте следния пример:

```
#!/bin/bash
x=5 # stojnosta na x e 5
# izpolzwame dwojni kawichki
echo "Using double quotes, the value of x is: $x"
# izpolzwame edinichni kawichki
echo 'Using forward quotes, the value of x is: $x'
```

Виждате ли разликата? Може да използвате двойни кавички, ако не смятате да слагате и променливи в низа, който ще заграждат кавичките. Ако се чудите дали единичните кавички запазват шпациите в даден низ, както това правят двойните кавички, погледнете следния пример:

```
xconsole$ mkdir 'hello world'
xconsole$ ls -F
hello world/
```

Обратно наклонените кавички коренно се различават от двойните и единични кавички. Те не се използват, за да запазват шпациите. Ако си спомняте по-рано използвахме следния ред:

```
x=$(expr $x + 1)
```

Както вече знаете, резултатът от командата expr $x + 1 се присвоява на променливата x. Същият този резултат може да бъде постигнат ако използваме обратно наклонени кавички:

```
x=`expr $x + 1`
```

Кой от начините да използвате? Този, който предпочитате. Ще откриете, че обратно наклонените кавички са по-често използвани от $(...). В много случаи обаче $(...) прави кода по-лесен за четене. Вземете предвид това:

```
$!/bin/bash
echo "I am `whoami`"
```

Аритметически операции с BASH
bash ви позволява да смятате различни аритметични изрази. Както вече видяхте аритметическите операции се използват посредством командата expr. Тази команда, обаче, както и командата true се смята, че са доста бавни. Причината за това е, че шел интерпретаторът трябва всеки път да стартира true и expr командите, за да ги използва. Като алтернатива на true посочихме командата ":". А като алтернатива на expr ще изпозваме следния израз $((...)). Разликата със $(...) е броят на скобите. Нека да опитаме:

```
#!/bin/bash
x=8 # stojnosta na x e 8
y=4 # stojnosta na y e 4
# sega shte priswojm sumata na promenliwite x i y na promenliwata z:
z=$(($x + $y))
echo "Sum na $x + $y e $z"
```

Ако се чуствате по-комфортно с expr вместо $((...)), тогава го използвайте него.

С bash можете да събирате, изваждате, умножавате, делите числа, както и да делите по модул. Ето и техните символи:

ДЕЙСТВИЕ	ОПЕРАТОР
Събиране	+
Изваждане	-
Умножение	*
Деление	/
Деление по модул	%

Всеки от Вас би трябвало да знае какво правят първите четири оператора. Ако не знаете какво означава деление по модул това е остатъкът при деление на две стойности. Ето и малко bash аритметика:

```
#!/bin/bash
x=5 # initialize x to 5
y=3 # initialize y to 3

add=$(($x + $y)) # sumiraj x sys y i priswoj rezultata na promenliwata add
sub=$(($x - $y)) # izwadi ot x y i priswoj rezultata na promenliwata sub
mul=$(($x * $y)) # umnozhi x po y i priswoj rezultata na promenliwata mul
div=$(($x / $y)) # razdeli x na y i priswoj rezultata na promenliwata div
mod=$(($x % $y)) # priswoj ostatyka pri delenie na x / y na promenliwata mod

# otpechataj otgoworite:
echo "Suma: $add"
echo "Razlika: $sub"
echo "Projzwedenie: $mul"
echo "Quotient: $div"
echo "Ostatyk: $mod"
```

Отново горният код можеше да бъде написан с командата expr. Например вместо add=$(($x + $y)), щяхме да пишем add=$(expr $x + $y), или add=`expr $x + $y`.

Четене на информация от клавиатурата
Сега вече идваме към интересната част. Вие можете да направите Вашите програми да си взаимодействат с потребителя и потребителят да може да си взаимодейства с програмата. Командата която Ви позволява да прочетете каква стойност в въвел потребителят е read. read е вградена в bash команда, която се използва съвместно с променливи, както ще видите:

```
#!/bin/bash
# gets the name of the user and prints a greeting
echo -n "Enter your name: "
read user_name
echo "Hello $user_name!"
```

Променливата тук е user_name. Разбира се, може да я наречете, както си искате. read ще ви изчака да въведето нещо и да натиснете клавиша ENTER. Ако не натиснете нищо, командата read ще чака, докато натиснете ENTER . Ако ENTER е натиснат, без да е въведено нещо, то ще продължи изпълнението на програмата от следващия ред. Ето и пример:

```
#!/bin/bash
# gets the name of the user and prints a greeting
echo -n "Enter your name: "
read user_name

# the user did not enter anything:
if [ -z "$user_name" ]; then
echo "You did not tell me your name!"
exit
fi

echo "Hello $user_name!"
```

Ако потребителят натисне само клавиша ENTER, нашата програма ще се оплаче и ще прекрати изпълнението си. В противен случай ще изпечата поздравление. Четенето на информацията, която се въвежда от клавиатурата е полезно, когато правите интерактивни програми, които изискват потребителят да отговори на конкретни въпроси.

Функции
Функциите правят скрипта по-лесен за поддържане. Най-общо казано, функциите разделят програмата на малки части. Функциите изпълняват действия, които Вие сте дефинирали и може да върне стойност от изпълнението си ако желаете. Преди да продължим, ще Ви покажа един пример на шел програма, която използва функция:

```
#!/bin/bash

# functiqta hello() samo izpechatwa syobshtenie
hello()
{
echo "Wie ste wyw funkciq hello()"
}

echo "Izwikwame funkciqta hello()..."
# izwikwame hello() funkciqta wytre w shell skripta:
hello
echo "Weche izleznahte ot funkciqta hello()"
```

Опитайте се да напишете тази програма и да я стартирате. Единствената цел на функцията hello() е да изпечата съобщение. Функциите естествено могат да изпълняват и по-сложни задачи. В горния пример ние извикахме функцията hello() с този ред:

```
hello
```

Когато се изпълнява този ред bash интерпретатора претърсва скрипта за ред който започва с hello(). След което открива този ред и изпълнява съдържанието на функцията.

Функциите винаги се извикват чрез тяхното име. Когато пишете функция можете да започнете функцията с function_name(), както беше направено в горния пример, или да използвате думата function т.е function

function_name(). Другият начин, по който можем да започнем нашата функция е function hello():

```
function hello()
{
echo "Wie ste wyw funkciq hello()"
}
```

Функциите винаги започват с отваряща и затваряща скоба"()", последвани от отварящи и затварящи къдрави скоби: "{...}". Тези къдрави скоби бележат началото и края на функцията. Всеки ред с код затворен в тези скоби ще бъде изпълнен и ще принадлежи единствено на функцията. Функциите трябва винаги да бъдат дефинирани преди да бъдат извикани. Нека погледнем нашата програма, само че този път ще извикаме функцията преди да е дефинирана:

```
#!/bin/bash
echo "Izwikwame funkciqta hello()..."
# call the hello() function:
hello
echo "Weche izleznahte ot funkciqta hello()"

# function hello() just prints a message
hello()
{
echo "Wie ste wyw funkciq hello()"
}
```

Ето какъв е резултатът, когато се опитаме да изпълним програмата:

```
xconsole$ ./hello.sh
Izwikwame funkciqta hello()...
./hello.sh: hello: command not found
Weche izleznahte ot funkciqta hello()
```

Както виждате, програмата върна грешка. Ето защо е добре да пишете Вашите функции в началото на скрипта или поне преди си ги извикате. Ето друг пример как да използваме функции:

```
#!/bin/bash
# admin.sh - administrative tool

# function new_user() creates a new user account
new_user()
{
echo "Preparing to add a new user..."
sleep 2
adduser # run the adduser program
}

echo "1. Add user"
echo "2. Exit"

echo "Enter your choice: "
read choice

case $choice in
1) new_user # call the new_user() function
;;
*) exit
;;
esac
```

За да работи правилно тази програма, трябва да сте влезли като root, тъй като adduser е програма, която само root потребителят има право да изпълнява. Да се надяваме, че този кратък пример Ви е убедил в полезноста на фукциите.

Прихващане на сигнали
Може да използвате вградената команда trap, за да прихващате сигнали във Вашата програма. Това е добър начин да излезете нормално от програмата. Непример, ако имате вървяща програма при натискането на CTRL-C ще изпратите на програмата interrupt сигнал, който ще "убие" програмата. trap ще Ви позволи да прихване този сигнал и ще Ви даде възможност или да продължите с изпълнението на програмата, или да съобщите на потребителя, че програмата спира изпълнението си. trap има следния синтаксис:

trap dejstwie signal

dejstwie указва какво да искате да направите, когато прихванете даден сигнал, а signal е сигналът, който очакваме. Списък със сигналите може да откриете като пишете trap -l. Когато използвате сигнали във вашата шел програма пропуснете първите три букви на сигнала, обикновено те са SIG. Например, ако сигналът за прекъсване е SIGINT, във Вашата шел програма използвайте само INT. Можете да използвате и номера на сигнала. Номерът на сигнала SIGINT е 2. Пробвайте следната програма:

```
#!/bin/bash
# using the trap command

# da se zipylni funkciqta sorry() pri natiskane na CTRL-C:
trap sorry INT

# function sorry() prints a message
sorry()
{
echo "I'm sorry Dave. I can't do that."
sleep 3
}

# count down from 10 to 1:
for i in 10 9 8 7 6 5 4 3 2 1; do
echo $i seconds until system failure."
sleep 1
done
echo "System failure."
```

Сега, докато програмата върви и брои числа в обратен ред натиснете CTRL-C. Това ще изпрати сигнал за прекъсване на програмата. Сигналът ще бъде прихванат от trap командата, която ще изпълни sorry() функцията. Можете да накарате trap да игнорира сигнал като поставите """ на мястото на действие. Можете да накарате trap да не прихваща сигнал като използвате "-". Например:

```
# izpylni sorry() funkciqta kogato programa poluchi signal SIGINT:
trap sorry INT

# nakaraj programata da NE prihwashta SIGINT signala :
trap - INT

# ne prawi nishto kogato se prihwane signal SIGINT:
trap " INT
```

Когато кажете на trap да не прихваща сигнала, то програмата се подчинява на основното действие на сигнал, което в конкретния случай е да прекъсне програмата и да я "убие". Когато укажете trap да не прави нищо при получаване на конкретен сигнал, то програмата ще продължи своето действие, игнорирайки сигнала.

Писане на скриптове за BASH шел : версия 1.2(част 4)
от X_console(6-06-2000)

Тази статия е преведена с разрешението на автора и X_console.
Адресът на оригиналната статия е http://xfactor.itec.yorku.ca/~xconsole/.

Продължение на [част 3].

AND и OR
Видяхме как се използват условните оператори и колко полезни са те. Има две допълнителни неща, които могат да бъдат добавени. Условните изрази са AND (или "&&") и OR (или "||"). AND условният израз изглежда по следния начин:

условие_1 && условие_2

AND изразът проверява първо най-лявото условие. Ако условието е вярно се проверява второто условие. Ако и то е вярно се изпълнява останалата част от кода на скрипта. Ако условие условие_1 не е вярно(върне резултат false), тогава условие условие_2 няма да бъде проверено. С други думи:

if(ако) условие_1 е вярно, AND(и) if(ако) условие_2 е вярно, then(тогава)...

Ето един пример с AND условие:

```
#!/bin/bash
x=5
y=10
if [ "$x" -eq 5 ] && [ "$y" -eq 10 ]; then
echo "I dwete uslowiq sa wqrni."
else
echo "Uslowiqta ne sa wqrni."
fi
```

Тук виждаме, че x и y имат стойността, за която проверяваме. Променете стойноста на x от x=5 на x=12, след което пуснете отново програмата и ще се убедите, че условието не е изпълнено(връща стойност false).

OR изразът е подобен. Единствената разлика е, че проверява дали най-левият израз не е верен(т.е връща резултат false). Ако това е изпълнено се проверява следващия израз и по-следващия:

условие_1 || условие_2

С други думи това звучи така:

if(ако) условие_1 е вярно, OR(или) ако условие_2 е вярно, тогава...

Ето защо кодът след този условен оператор ще бъде изпълнен ако поне едно от условията е вярно:

```
#!/bin/bash
x=3
y=2
if [ "$x" -eq 5 ] || [ "$y" -eq 2 ]; then
echo "Edno ot uslowiqta e wqrno."
else
echo "Nito edno ot uslowiqta ne e wqrno."
fi
```

В този пример ще се уверите, че едно от условията е вярно. Сменете стойността на променливата y и изпълнете отново програмата. Ще видите, че нито едно от условията не е вярно.

Ако се замислите, ще видите, че условният оператор if може да замести употребата на AND и OR изразите. Това става чрез използването на вложени if оператори. "Влагане на if оператори" означава да използваме if оператор в тялото на друг if оператор. Можете да правите влагане и на други оператори, а не само на if. Ето един пример с вложени ifоператори, които заместват използването на AND израз в кода на програмата:

```
#!/bin/bash
x=5
y=10
if [ "$x" -eq 5 ]; then
if [ "$y" -eq 10 ]; then
echo "I dwete uslowiq sa wqrni."
else
echo "Uslowiqta ne sa wqrni."
fi
fi
```

Резултатът е същият, както и ако използвахме AND израз. Проблемът е, че кодът става по - трудно четим и отнема повече време, за да се напише. За да се предпазите от проблеми използвайте AND и OR изрази.

Използване на аргументи
Може би сте забелязали, че повечето програми в Linux не са интерактивни. От Вас се иска да въведете някакви аргументи; в противен случай получавате съобщение, в което се обяснява как да използвате програмата. Вземете за пример командата more. Ако не напишете име на файл след нея, резултатът ще бъде точно едно такова помощно съобщение. Възможно е да направите вашата шел програма да използва аргументи. За тази цел трябва да използвате специалната променлива "$#". Тази променлива съдържа общия брой на всички аргументи подадени на програмата. Например ако изпълните следната програма:

```
xconsole$ foo argument
```

$# ще има стойност 1 , защото има само един аргумент подаден на програмата. Ако имате два аргумента, тогава $# ще има стойност 2. В допълнение стойността на всеки аргумент (нулевият аргумент е винаги името на програма - foo) може да се вземе като използвате променливите $0 - за името на програмата в случая foo, $1 за стойноста на първият аргумент -argument и т.н. Може да имате максимум 9 такива променливи от $0 до $9. Нека да видим това в действие:

```
#!/bin/bash
# izpechataj pyrwiq argument
# proweri dali ima pone edin argument:
if [ "$#" -ne 1 ]; then
echo "usage: $0 "
fi

echo "Stojnosta na argumenta e $1"
```

Тази програма очаква един и само един аргумент, за да тръгне. Ако я стартирате без аргументи, или подадете повече от един аргумент, програмата ще изпечата съобщение за това как да се използва. В случай, че имаме само един аргумент шел програмата ще отпечата стойноста на аргумента, който сте подали. Припомнете си, че $0 е името на програмата. Ето защо тази специална променлива се използва в "usage" съобщението.

Пренасочване и PIPING
Обикновено, когато стартирате дадена команда, резултатът от изпълнението се отпечатва на екрана. Например:

```
xconsole$ echo "Hello World"
Hello World
```

"Пренасочването" Ви позволява да съхраните резултата от изпълнението някъде другаде. В повечето случаи това става към файл. Операторът ">" се използва за пренасочване на изхода. Мислете за него като за стрелка, сочеща къде да отиде резултата. Ето един пример за пренасочване на изхода към файл:

```
xconsole$ echo "Hello World" > foo.file
xconsole$ cat foo.file
Hello World
```

Тук резултатът от командата echo "Hello World" е пренасочен към файл с име foo.file. Когато прочетете съдържанието на файла ще видите там резултата. Има един проблем, когато използвате оператора ">". Ако имате файл със същото име, то неговото съдържание няма да бъде запазено, а ще бъде изтрито и заместено с новото. Ами ако искате да добавите информация във файла, без да изтривате старата? Тогава трябва да използвате операторът за добавяне : ">>". Използва се по същия начин с тази разлика, че не изтрива старото съдържание на файла, а го запазва и добавя новото съдържание накрая.

А сега ще ви запознаем с piping. Piping-ът Ви позволява да вземете резултата от изпълнението на дадена програма и да го използвате като входни данни за друга програма. Piping става посредством оператора: "|". Забележете, че това не е малката буквата "L". Този оператор може да получите чрез натискане на клавиша SHIFT и \. Ето и един пример за piping:

```
xconsole$ cat /etc/passwd | grep xconsole
xconsole:x:1002:100:X_console,,,:/home/xconsole:/bin/bash
```

Тук четем целия файл /etc/passwd и след това резултатът е подаден за обработка на командата grep, която от своя страна претърсва текста за низа xconsole и изпечатва целия ред, съдържащ този низ на екрана. Може да използвате и пренасочване, за да запишете крайния резултат на файл:

```
xconsole$ cat /etc/passwd | grep xconsole > foo.file
xconsole$ cat foo.file
xconsole:x:1002:100:X_console,,,:/home/xconsole:/bin/bash
```

Работи. Файлът /etc/passwd е прочетен и неговото съдържание е претърсено от командата grep за низа xconsole. След което крайният резултат е пренасочен към файл foo.file. Ще откриете, че пренасочване и piping са много полезни средства, когато пишете Вашите шел програми.

Временни файлове
Често ще има моменти, в които ще Ви се наложи да създадете временен файл. Този файл може да съдържа временна информация и просто да работи с някоя програма. В повечето случаи със завършването на изпълнението на програмата се изтрива и временният файл. Когато създадете файл трябва да му зададете име. Проблемът е, че името на файла, който създавате не трябва да съществува в директорията, в която го създавате. В противен случай може да затриете важна информация. За да създадете файл с уникално име трябва да използвате "$$" символа като представка или надставка в името на файла. Вземете за пример следния случай: искате да създадете временен файл с име hello. Има вероятност и някой друг да има файл със същото име в тази директория, което ще доведе до катастрофални резултати за Вашата програма. Ако вместо това създадете фиайл с име hello.$$ или $$hello, Вие ще създадете уникален файл. Опитайте:

```
xconsole$ touch hello
xconsole$ ls
hello
xconsole$ touch hello.$$
xconsole$ ls
hello hello.689
```

Ето го и нашият временен файл.

Връщане на стойности
Повечето програми връщат стойност(и) в зависимост от начина, по който завършват изпълнението си. Например, ако разгледате ръководството на командата grep ще видите, че в него се казва, че командата grep връща стойност 0 ако има съвпадение, и 1 ако не е открито съвпадение. Защо да се грижим да връщаме стойности? По много причини. Нека да кажем, че искате да проверите дали конкретно потребителско име съществува на Вашата система. Единият от начините да направите това е да използвате командата grep върху файла с паролите /etc/passwd. Да предположим, че потребителското име, което търсим е foobar:

```
xconsole$ grep "foobar" /etc/passwd
xconsole$
```

Няма никакъв резултат от изпълнението. Това означава че grep не е намерила съвпадение. Но може да направим програмата много по-полезна ако се появява съобщение, което пояснява резултата. Това е, когато искате да проверите стойноста, която се връща от дадена програма. Има една специална променлива, която съдържа крайния резултат от изпълнението на програмата. Тази променлива е $?.Разгледайте следния код:

```
#!/bin/bash
# grep for user foobar and pipe all output to /dev/null:
grep "foobar" /etc/passwd > /dev/null 2>&1
# capture the return value and act accordingly:
if [ "$?" -eq 0 ]; then
echo "Match found."
exit
else
echo "No match found."
fi
```

Когато стартираме програмата, променливата "$?" ще прихване резултата от командата grep. Ако той е равен на 0, значи има съвпадение и подходящо съобщение ще обяви за това. В противен случай ще изпечата, че няма съвпадения. Това е един основен начин за получаване на резултата, който връща дадена програма. Ще откриете, че доста често ще Ви се наложи да знаете стойността, която връща дадена програма, за да продължите по-нататък.

Ако случайно се чудите какво значи 2>&1, сега ще ви обясня . Под Linux, тези номера обозначават файлови дескриптори. 0 е за стандартния вход (пример: клавиатура), 1 е за стандартния изход (пример: монитор) и 2 е за стандартния изход на грешките (пример: монитор). Всяка обикновена информация се изпраща на файлов дескриптор 1, и ако има грешки те се изпращат на файлов дескриптор 2. Ако не искате тези съобщения да излизат просто можете да ги пренасочите към /dev/null. Забележете, че това няма да спре изпращането на информацията на стандартния изход. Например, ако нямате права да четете от директория на друг потребител, Вие няма да можете да видите нейното съдържание:

```
xconsole$ ls /root
ls: /root: Permission denied
xconsole$ ls /root 2> /dev/null
xconsole$
```

Както виждате, съобщението за грешка не беше изпечатано. Същото важи както за други програми, така и за файлов дескриптор 1. Ако не искате резултатът от изпълнението на програмата да се отпечатва на екрана, можете спокойно да го пренасочите към /dev/null. Ако не искате да виждате както резултатът от изпълнението, така и съобщенията за грешка, може да го направите по следния начин:

```
xconsole$ ls /root > /dev/null 2>&1
```

Това означава че резултатът от програмата, както и всяка грешка, която предизвика тази програма ще бъдат изпратени на /dev/null, така че никога повече няма да можете да ги видите.

Какво трябва да направите ако искате Вашият шел скрипт да връща стойност при завършване на програмата? Командата exit приема само един аргумент - число, което трябва да се върне при завършване на програмата. Обикновно числото 0 се използва, за да кажем, че програмата е завършила успешно, т.е. не е възникнала никаква грешка по време на нейното изпълнение. Всичко по-голямо или по-малко от 0 обикновено обозначава, че е възникнала някаква грешка. Това го решавате Вие като програмист. Нека проследим следната програма:

```
#!/bin/bash
if [ -f "/etc/passwd" ]; then
echo "Password file exists."
exit 0
```

```
else
echo "No such file."
exit 1
fi
```

Заключение

С това завършихме увода в bash програмирането. Това ръководство Ви дава основните знания, за да можете да редактирате чужди bash скриптове или да създавате нови. За да постигнете съвършенство обаче, трябва много да практикувате. bash е идеално средство за писане на обикновени административни скриптове. Но за по-големи разработки ще се нуждаете от мощни езици като C или Perl.

МОЖЕТЕ ДА ДОПИСВАТЕ, КОРИГИРАТЕ УЧЕБНИКА :)